UNIVERZA V LJUBLJANI
FAKULTETA ZA RAČUNALNIŠTVO IN INFORMATIKO

Polona Bogataj

Faktorizacija naravnih števil

DIPLOMSKO DELO
NA INTERDISCIPLINARNEM UNIVERZITETNEM ŠTUDIJU

Mentor: prof. dr. Borut Robič

Ljubljana, 2011

Univerza
v Ljubljani

Fakulteta za računalništvo
in informatiko

Tržaška 25
1000 Ljubljana, Slovenija
telefon: 01 476 84 11
faks: 01 426 46 47
www.fri.uni-lj.si
e-mail: dekanat@fri.uni-lj.si

Št. naloge: 00025/2011
Datum: 15.03.2011

Univerza v Ljubljani, Fakulteta za računalništvo in informatiko ter Fakulteta za matematiko in fiziko izdaja naslednjo nalogo:

Kandidat: **POLONA BOGATAJ**

Naslov: **FAKTORIZACIJA NARAVNIH ŠTEVIL**
INTEGER FACTORIZATION ALGORITHMS

Vrsta naloge: Diplomsko delo univerzitetnega študija

Tematika naloge:

Predstavite algoritme za faktorizacijo naravnih števil. Algoritme opišite, predstavite njihovo časovno zahtevnost in psevdokodo. Primerjajte posamezne algoritme in ugotovite, kdaj je njihova uporaba primerna. Izbrane algoritme implementirajte. Opišite uporabo algoritmov v računalništvu.

Mentor:

prof. dr. Borut Robič

Fakulteta za računalništvo in informatiko • UNIVERZA V LJUBLJANI • 2

Dekan Fakultete za računalništvo in informatiko:

prof. dr. Nikolaj Zimic

Dekan Fakultete za matematiko in fiziko:

prof. dr. Andrej Likar

UNIVERZA V LJUBLJANI • Fakulteta za matematiko in fiziko • 6

Rezultati diplomskega dela so intelektualna lastnina Fakultete za računalništvo in informatiko Univerze v Ljubljani. Za objavljanje ali izkoriščanje rezultatov diplomskega dela je potrebno pisno soglasje Fakultete za računalništvo in informatiko ter mentorja.

IZJAVA O AVTORSTVU

diplomskega dela

Spodaj podpisana Polona Bogataj,

z vpisno številko 63050182,

sem avtorica diplomskega dela z naslovom:

Faktorizacija naravnih števil

S svojim podpisom zagotavljam, da:

- sem diplomsko delo izdelala samostojno pod mentorstvom
  prof. dr. Boruta Robiča
- so elektronska oblika diplomskega dela, naslov (slov., angl.), povzetek (slov., angl.) ter ključne besede (slov., angl.) identični s tiskano obliko diplomskega dela
- soglašam z javno objavo elektronske oblike diplomskega dela v zbirki "Dela FRI".

V Ljubljani, dne 16.5.2011 Podpis avtorice:

Zahvala

V prvi vrsti se zahvaljujem prof. dr. Borutu Robiču za pomoč in usmerjanje pri izdelavi diplomske naloge.

Zahvala gre tudi moji družini, fantu in prijateljem za vse potrpljenje in spodbudo v času študija.

Kazalo

Povzetek **1**

Abstract **3**

1 Uvod **5**
1.1 Uporaba faktorizacije 8

2 Namenski algoritmi **10**
2.1 Poskusno deljenje 10
2.2 Faktorizacija v algebraičnih grupah 13
2.2.1 Pollardov $p-1$ algoritem 13
2.2.2 Williamsov $p+1$ algoritem 15
2.2.3 Lenstrova metoda (ECM) 19
2.3 Pollardov ρ algoritem 24
2.4 Fermatov algoritem 26
2.5 Eulerjev algoritem 28

3 Splošni algoritmi **33**
3.1 Dixonov algoritem 33
3.2 Kvadratno sito (QS) 34
3.2.1 Kvadratno sito na družini polinomov (MPQS) 42
3.3 Faktorizacija z verižnimi ulomki 43
3.3.1 Verižni ulomki 43
3.3.2 Algoritem za faktorizacijo z verižnimi ulomki (CFRAC) 46
3.3.3 Shanksova faktorizacija s kvadratnimi formami (SQUFOF) 48
3.4 Številsko sito (NFS) 52
3.4.1 Primerjava algoritmov NFS in QS 59
3.4.2 Posebno številsko sito (SNFS) 59

4 Zaključek **61**

A Implementacija algoritmov **63**

Stvarno kazalo **73**

Seznam algoritmov **74**

Literatura **75**

Seznam uporabljenih kratic in simbolov

- *Trial division* - Poskusno deljenje
- *Pollard's* $p-1$ *algorithm* - Pollardov $p-1$ algoritem
- *Williams's* $p+1$ *algorithm* - Williamsov $p+1$ algoritem
- *Lenstra elliptic curve factorization*: ECM - Lenstrova metoda
- *Pollard's* ρ *algorithm* - Pollardov ρ algoritem
- *Fermat's factorization method* - Fermatov algoritem
- *Euler's factorization method* - Eulerjev algoritem
- *Dixon's algorithm* - Dixonov algoritem
- *Quadratic sieve*: QS - Kvadratno sito
- *Multiple Polynomial Quadratic Sieve*: MPQS - Kvadratno sito na družini polinomov
- *Continued fraction factorization*: CFRAC - Algoritem za faktorizacijo z verižnimi ulomki
- *Shanks' square forms factorization*: SQUFOF - Shanksova faktorizacija s kvadratnimi formami
- *Number field sieve*: NFS - Številsko sito
- *Special number field sieve*: SNFS - Posebno številsko sito
- *Shor's algorithm* - Shorov algoritem

Povzetek

Razcep naravnega števila na zmnožek praštevil imenujemo faktorizacija. Problem faktorizacije je zanimiv zato, ker ne poznamo učinkovitega algoritma, s katerim bi v polinomskem času faktorizirali podano naravno število n. Najbližje temu cilju je Shorov algoritem za kvantne računalnike, vendar pa še ni uporaben v praksi. Prav na težavnosti problema faktorizacije temeljijo moderni kriptosistemi, najbolj poznan med njimi je RSA.

Namen diplomskega dela je predstavitev različnih algoritmov za faktorizacijo naravnih števil. Za vsak algoritem sta poleg opisa podani tudi njegova časovna zahtevnost in psevdokoda, nekateri algoritmi pa so tudi implementirani v programskem jeziku Java.

Diplomsko delo je razdeljeno v več delov. V prvem delu je predstavljeno matematično ozadje ter uporaba faktorizacije. Drugi del je namenjen ti. namenskim algoritmom, ki imajo časovno zahtevnost odvisno od lastnosti števila, ki ga želimo faktorizirati. Spoznali bomo algoritem poskusno deljenje, Pollardova $p - 1$ in ρ algoritma, Williamsov $p + 1$ algoritem, Lenstrovo metodo, Fermatovo metodo in Eulerjevo metodo. Tretji del pa je namenjen splošnim algoritmom, ki imajo časovno zahtevnost odvisno le od velikosti števila, ki ga želimo faktorizirati. Mednje spadajo Dixonov algoritem, kvadratno in številsko sito, algoritem za faktorizacijo z verižnimi ulomki in Shanksova faktorizacija s kvadratnimi formami. V zaključku je povzetek algoritmov s časovnimi zahtevnostmi in smernice za nadaljno delo.

Ključne besede:

faktorizacija, algoritem, časovna zahtevnost, praštevilo, največji skupni delitelj

Abstract

The decomposition of a natural number into a product of prime numbers is called factorization. The main problem with factorization is the fact that there is no known efficient algorithm which would factor a given natural number n in polynomial time. The closest equivalent to such an algorithm is Shor's algorithm for quantum computers, which is still not practically applicable. The difficulties with factorization form the basis for modern cryptosystems—the most renowned among them is the RSA algorithm.

The purpose of this thesis is to present different algorithms for natural number factorization. For each algorithm, the thesis provides its description, its time complexity and its pseudocode. Some of the algorithms are implemented in the Java Programming Language.

The thesis is divided into several parts. The first part describes the mathematical characteristics and the use of factorization. The second part deals with special-purpose algorithms, whose time complexity depends on the properties of the factorized number. The algorithms presented in this part include trial division, Pollard's $p - 1$ and ρ algorithms, Williams' $p + 1$ algorithm, Lenstra elliptic curve factorization, Fermat's factorization method, and Euler's factorization method. The third part deals with general-purpose algorithms, whose time complexity depends solely on the size of the factorized number. These include Dixon's algorithm, the quadratic and number field sieves, the continued fraction factorization, and Shanks' square forms factorization. The conclusion consists of the summaries of the presented algorithms with their time complexities, and provides the guidelines for further research.

Key words:

factorization, algorithm, time complexity, prime number, greatest common divisor

Poglavje 1

Uvod

Naravna števila, ki so večja od 1, lahko razdelimo v dve skupini: praštevila in sestavljena števila. **Sestavljena števila** so tista števila, ki jih lahko zapišemo kot produkt dveh naravnih števil večjih od 1. **Praštevila** pa so tista števila, ki so deljiva le s številom 1 in samim seboj.

Definicija 1.1 (Faktorizacija)**.** Razcep naravnega števila na zmnožek praštevil imenujemo **faktorizacija**. Faktorizacijo števila n zapišemo kot:

$$n = p_1^{a_1} \times p_2^{a_2} \times \ldots p_r^{a_r},$$

kjer so $p_1, p_2, \ldots, p_r$ različna praštevila, $a_1, a_2, \ldots, a_r \in \mathbb{N}$.

Če dodamo urejenost praštevil ($p_1 < p_2 < \ldots < p_r$), dobimo enolično faktorizacijo.

V splošnem delimo algoritme za faktorizacijo v dve skupini - namenske algoritme in splošne algoritme. Ločimo jih glede na časovno zahtevnost - v prvi skupini so algoritmi, katerih časovna zahtevnost je odvisna od lastnosti števila, ki ga želimo faktorizirati. Za algoritme, ki smo jih uvrstili med splošne algoritme pa velja, da je njihova časovna zahtevnost odvisna le od velikosti števila, ki ga želimo faktorizirati.

Definicija 1.2 (Tuji števili)**.** Če naravni števili m in n nimata skupnega praštevila v faktorizaciji posameznega števila, potem pravimo, da števili m in n **tuji števili**.

V nadaljevanju bomo pogosto srečali gladka števila, zato na tem mestu podajmo njihovo definicijo.

Definicija 1.3 (Gladko število)**.** Naravno število n je **B-gladko število**, če noben izmed praštevilskih faktorjev v njegovi faktorizaciji ni večji od števila B.

Če zapišemo drugače: število $n = p_1^{a_1} \times p_2^{a_2} \times \ldots \times p_r^{a_r}$ je B-gladko natanko tedaj, ko velja $p_i \leq B, i = 1, 2 \ldots r$.

Definicija 1.4 (Največji skupni delitelj)**.** Za podani naravni števili a in b je njun **največji skupni delitelj** največje naravno število, ki deli a in b. Njegova oznaka je $\gcd(a, b)$.

Evklidov algoritem

Za izračun največjega skupnega delitelja se uporablja **Evklidov algoritem**. Oglejmo si razširjen Evklidov algoritem, ki poleg največjega skupnega delitelja števil a in b poda tudi vrednost b^{-1}.

Algoritem 1 Razširjen Evklidov algoritem

Vhod: števili a in b
Izhod: (r, s, t), kjer je največji skupni večkratnik $\gcd(a, b) = r$ in multiplikativni inverz $b^{-1} \equiv t \pmod{a}$

1: $a_0 = a, \quad b_0 = b$
2: $t_0 = 0, \quad t = 1$
3: $s_0 = 1, \quad s_1 = 0$
4: $q = \lfloor \frac{a_0}{b_0} \rfloor$
5: $r = a_0 + qb_0$
6: **while** $r > 0$ **do**
7: tmp$= t_0 - qt$
8: $t_0 = t$
9: $t =$tmp
10: tmp$= s_0 - qs$
11: $s_0 = s$
12: $s =$tmp
13: $a_0 = b_0$
14: $b_0 = r$
15: $q = \lfloor \frac{a_0}{b_0} \rfloor$
16: $r = a_0 - qb_0$
17: **end while**
18: $r = b_0$
19: **return** (r, s, t)

Časovna zahtevnost algoritma je $\mathcal{O}\left((\log n)^3\right)$.

Eratostenovo sito

Eratostenovo sito je preprost algoritem za iskanje praštevil, manjših od števila n. Najprej na papir zapišemo vsa števila od 2 do izbranega števila n. Prvo število (število 2) mora biti praštevilo. Po celotnem seznamu nato prečrtamo vse njegove večkratnike, število 2 pa pustimo neprečrtano. Naslednje neprečrtano število v našem seznamu je zopet praštevilo - v našem primeru je to število 3. Ponovno prečrtamo vse njegove večkratnike, ki so večji od njega, in nadaljujemo postopek na naslednjem neprečrtanem številu.

Algoritem 2 Eratostenovo sito

Vhod: število n
Izhod: praštevila, ki so manjša ali enaka n

for $2 \leq i \leq n$ **do**
    $a_i = i$
end for
$j = 2$
while $j^2 \leq n$ **do**
    if $a_j \neq 0$ **then**
        $t = 2j$
        while $t \leq n$ **do**
            $a_t = 0$
            $t = t + j$
        end while
    end if
end while
$\mathcal{P} = \{\}$
for $2 \leq i \leq n$ **do**
    if $a_i \neq 0$ **then**
        $\mathcal{P} = \mathcal{P} \cup \{i\}$
    end if
end for
return $\mathcal{P}$

Eratosten je opazil, da nam ni potrebno pregledati vseh števil do števila n, ampak le števila od 2 do $\sqrt{n}$. Ko najdemo praštevilo, ki je večje od $\sqrt{n}$, vemo, da so vsa neprečrtana števila, ki so večja od njega, praštevila. Če bi

bilo katero izmed teh števil sestavljeno število, bi moral imeti faktor manjši od svojega kvadratnega korena, in bi ga že prečrtali.

Primer. *Naj bo naše izbrano število* $n = 11$*. Njegov kvadratni koren je* $\sqrt{11} = 3,3$*, zato bomo pregledali le števila do prvega praštevila, ki je večji od* 3*.*

2	3	***4***	5	***6***		2	***3***	4	5	***6***		2	3	4	***5***	6
7	***8***	9	***10***	11		7	8	***9***	10	11		7	8	9	***10***	11

Tako smo našli vsa praštevila, ki so manjša ali enaka 11*:* $2, 3, 5, 7$ *in* 11*.*

Eratostenovo sito nam poda še eno uporabno informacijo: za vsako sestavljeno število lahko ugotovimo, koliko različnih praštevilskih faktorjev ima. V našem primeru ugotovimo, da imata števili 6 in 10 dva različna praštevilska faktorja. Na takšen način lahko identificiramo števila, ki imajo veliko majhnih praštevilskih faktorjev.

Definicija 1.5 (Eulerjeva funkcija φ)**.** Eulerjeva funkcija φ je multiplikativna aritmetična funkcija poljubnega naravnega števila n in da skupno število naravnih števil, ki ne presegajo n in so n tuja.

Za praštevilo p je funkcija definirana kot

$$\varphi(p) = p - 1.$$

Za tuji praštevili a in b pa je definirana kot

$$\varphi(a \cdot b) = \varphi(a) \cdot \varphi(b).$$

1.1 Uporaba faktorizacije

Gotovo se bralcu porodi vprašanje, ali je faktorizacija danes sploh aktualna. Izkaže se, da je zelo aktualna, saj prav na težavnosti problema faktorizacije temeljijo moderni kriptosistemi, najbolj poznan je **RSA**.

RSA spada med kriptosisteme z javnimi ključi, kar pomeni, da za njegovo uporabo ne potrebujemo varnega kanala. Udeležena oseba A ima zato javni in zasebni ključ.

Za generiranje ključev najprej izberemo praštevili p in q ter izračunamo modul n:

$$n = p \cdot q.$$

Nato izberemo šifrirni eksponent e, da velja

$$\gcd(e, \varphi(n))) = 1$$

in iz kongruence

$$e \cdot d \equiv 1 \pmod{\varphi(n)}$$

z razširjenim Evklidovim algoritmom izračunamo odšifrirni eksponent d. Javni ključ je par (e, n), zasebni ključ pa trojka (d, p, q).

Kako poteka šifriranje in dešifriranje? Šifriranje sporočila x:

$$E(e, n)(x) = x^e \pmod{n},$$

dešifriranje prejetega sporočila y pa poteka po naslednji enačbi:

$$D(d, p, q)(y) = y^d \pmod{n}.$$

Varnost sistema temelji na tajnosti uporabnikovega zasebnega ključa. Če bi napadalec uspel faktorizirati modul n, bi lahko izračunal $\varphi(n)$ in odšifrirni eksponent d ter bi lahko enostavno odšifriral vsa šifrirana sporočila uporabnika A.

V praksi to pomeni, da mora biti modul n dovolj velik, da ga ne moremo faktorizirati v realnem času. Danes je priporočena dolžina ključa 2048 bitov, saj znanstveniki domnevajo, da bo kmalu mogoče razbiti ključ dolžine 1024. Za daljnoročno varnost šifriranih podatkov znanstveniki priporočajo celo 4096-bitne ključe. Za primerjavo povejmo, da lahko RSA ključ dolžine 300 bitov enostavno razbijemo v nekaj urah na domačem računalniku.

Leta 1991 so v RSA laboratorijih objavili prvi **izziv RSA** - objavili so več velikih števil, ki imajo natanko dva faktorja. Najmanjše število izziva RSA je 100-mestno število, ki so ga uspešno faktorizirali že mesec dni po objavi. Največje število je 617-mestno število izziva RSA-2048, ki ga do danes še ni nihče uspel faktorizirati. Izziv se je zaključil leta 2007, a še do danes velja za merilo napredka faktorizacije in varnosti RSA ključev.

Leta 1994 je Peter Shor dokazal, da bi s kvantnim računalnikom v polinomskem času razbili RSA - več o Shorovem algoritmu si lahko preberemo v [Sh94]. V zadnjih letih je razvoj kvantnih računalnikov napredoval in tako so leta 2009 uspeli implementirati nepopoln Shorov algoritem na silikonski kvantni čip.

Poglavje 2

Namenski algoritmi

V tem poglavju bomo predstavili algoritme, za katere velja, da je časovna zahtevnost algoritma odvisna od lastnosti števila n, ki ga želimo faktorizirati.

Mednje sodijo naslednji algoritmi:

- poskusno deljenje
- algoritmi za faktorizacijo v algebraičnih grupah
  - Pollardov $p-1$ algoritem
  - Williamsov $p+1$ algoritem
  - Lenstrova metoda
- Pollardov ρ algoritem
- Fermatova metoda
- Eulerjeva metoda
- posebno številsko sito (algoritem je predstavljen v poglavju 3.4.2)

2.1 Poskusno deljenje

Poskusno deljenje je metoda, kjer zaporedno poskušamo deliti število n s poskusnimi delitelji in tako dobimo delno ali popolno faktorizacijo števila n. Algoritem je enostaven, vendar učinkovit le za števila, ki imajo relativno majhne praštevilske faktorje.

Začnemo s prvim praštevilom, številom 2, in z njim delimo n dokler je ostanek pri deljenju enak 0. Preostali nefaktorizirani del števila n poskusimo

deliti z naslednjim praštevilom. Postopek nadaljujemo dokler ne dosežemo poskusnega delitelja, ki je večji od korena preostalega nefaktoriziranega dela števila. Takrat se lahko ustavimo, saj je nefaktoriziran del števila praštevilo.

Primer. *Naj bo podano število* $n = 7399$*.*

Najprej poskusimo s praštevili 2*,* 3 *in* 5*, vendar nobeden izmed njih ne deli števila* n*. Naslednje praštevilo je* 7*:*

$$7399 \div 7 = 1057.$$

Preverimo, ali 7 *deli preostali nefaktorizirani del števila:*

$$1057 \div 7 = 151.$$

Ker 7 *ne deli* 151*, poskusimo z naslednjim poskusnim deliteljem, ki je število 11. Ugotovimo, da 11 ne deli 151. Ker je* $\sqrt{151} = 12,3$ *in je naslednji poskusni delitelj večji od njega, ugotovimo, da je 151 praštevilo.*

Tako je faktorizacija števila 7399*:* $7399 = 7^2 \times 151$*.*

Opisan algoritem zahteva, da imamo pripravljen seznam vseh praštevil, ki so manjša od $\sqrt{n}$. Ker je to precej zahtevna predpriprava na enkratno izvedbo algoritma, lahko za množico poskusnih deliteljev vzamemo kar vsa števila, saj sestavljena števila ne bodo vplivala na končno faktorizacijo. Poglejmo si primer:

Primer. *Naj bo* $n = 492$*.*

Kot prvi preizkusni delitelj nastopi število 2*:*

$$492 \div 2 = 246, \quad 246 \div 2 = 123.$$

Ker preostali nefaktoriziran del ni deljiv z 2*, poskusimo s* 3*:*

$$123 \div 3 = 41.$$

Ko poskusimo 41 *deliti s* 3*,* 4*,* 5 *in* 6 *ugotovimo, da nobeden ne deli števila* 41*. Ker je naslednji poskusni delitelj* 7 *večji kot* $\sqrt{41} \doteq 6,4$*, ugotovimo, da je* 41 *praštevilo.*

Tako lahko zapišemo faktorizacijo števila 492*:* $492 = 2^2 \times 3 \times 41$*. Na končno faktorizacijo tako ni vplivalo dejstvo, da smo pregledali vsa števila, ne le praštevila.*

Algoritem 3 Poskusno deljenje

Vhod: naravno število n
Izhod: množica $\mathcal{F}$, ki vsebuje praštevilske faktorje števila n

```
1: F={}                                   ▷ Množica praštevilskih faktorjev
2: N = n
3: while 2 | N do
4:     N = N/2
5:     F = F ∪ {2}
6: end while
7: d = 3
8: while d^2 ≤ N do
9:     while d | N do
10:        N = N/d
11:        F = F ∪ {d}
12:    end while
13:    d = d + 2
14: end while
15: if N == 1 then
16:    return F
17: else
18:    return F ∪ {N}
19: end if
```

Za majhno pohitritev lahko poskrbimo tako, da upoštevamo, da je 2 edino sodo praštevilo. Zato lahko poleg 2 pregledamo le liha števila. Tako poenostavljen algoritem je predstavljen kot Algoritem 3.

Če algoritem poženemo večkrat (ga recimo kličemo kot podprogram), je smotrno, da si pripravimo seznam praštevil. Enostavnejše, a še vedno učinkovito je, če za poskusne delitelje uporabimo števili 2 in 3 ter vsa pozitivna cela števila oblike $6k \pm 1$, kjer je $k > 0$. Tako zajamemo vsa praštevila ter tudi nekatera sestavljena števila (npr. $25, 35, 49, \ldots$).

Časovna zahtevnost

Kakšna je časovna zahtevnost algoritma 3? Pri izpeljavi si bomo pomagali s praštevilskim izrekom, ki nam pove, kakšna je gostota praštevil.

Izrek 2.1 (Praštevilski izrek)**.** *Naj bo* $\Pi(n)$ *število praštevil, ki so manjša ali enaka številu* n*, za poljubno realno število* n*. Funkcija* $\Pi(n)$ *je asimptotično*

enaka $\frac{x}{\ln x}$, *ko gre* $x \to \infty$.

Prvo poglejmo najslabši možni primer, ki se zgodi, ko je n praštevilo. Tedaj algoritem pregleda vse poskusne delitelje do $\sqrt{n}$. Če za poskusne delitelje uporabimo vsa liha števila, ki so manjša od $\sqrt{n}$ in število 2, potem pregledamo približno $\frac{1}{2}\sqrt{n}$ števil. Če pa uporabimo seznam praštevil, potem je po praštevilskem izreku (Izrek 2.1) število pregledanih števil (tj. praštevil manjših od $\sqrt{n}$) približno $\frac{2\sqrt{n}}{\ln n}$.

V primeru, ko je n zmnožek dveh velikih praštevil, je za faktorizacijo sestavljenega števila potrebnih $\sqrt{n}$ korakov. Kaj pa v povprečju? Izkaže se, da je tudi v splošnem časovna zahtevnost $\mathcal{O}(\sqrt{n})$, saj prevladajo števila z velikim praštevilskim faktorjem. Če izvzamemo 50% najslabših števil in izračunamo povprečno časovno zahtevnost algoritma za popolno faktorizacijo števil, dobimo časovno zahtevnost n^c, kjer je $c = \frac{1}{2\sqrt{e}} \doteq 0,303$.

Ko govorimo o velikosti števila n, imamo ponavadi v mislih število bitov števila n, npr. 1024, kar pomeni, da je število $n = 2^{1024} \approx 10^{308}$ in moramo v našem primeru pregledati števila do okoli 10^{154} (oz. 10^{151} če pregledamo le praštevila).

Z algoritmom poskusno deljenje lahko enostavno poiščemo majhne faktorje števila n. Pri tem lahko omenimo, da imamo pri naključno izbranem številu n 50% možnosti, da je deljivo z 2 in 33% možnosti, da je deljivo s 3. Da se pokazati, da ima 88% naravnih števil vsaj en faktor manjši kot 100 in 92% vsaj enega manjšega kot 1000.

2.2 Faktorizacija v algebraičnih grupah

Algoritmi za faktorizacijo v algebraičnih grupah faktorizirajo število n tako, da računajo v algebraičnih grupah, ki so definirane z modulom n (npr. končnih obsegih, eliptične krivulje, ...). Mednje uvrščamo naslednje algoritme:

- Pollardov $p-1$ algoritem,
- Williamsov $p+1$ algoritem in
- Lenstrova metoda, ki za algebraično grupo uporablja eliptično krivuljo.

2.2.1 Pollardov $p-1$ algoritem

Pollardov $p-1$ algoritem temelji na lastnosti, ki jo lahko izpeljemo iz malega Fermatovega izreka.

Izrek 2.2 (Mali Fermatov izrek)**.** *Naj bo p liho praštevilo. Potem velja:*

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

Naj ima število n, ki ga želimo faktorizirati, v svoji faktorizaciji praštevilski faktor p z lastnostjo: vsi praštevilski faktorji števila $p-1$ so majhna števila in $p-1$ deli 10000!. Precej hitro lahko izračunamo

$$m = 2^{10000!} \mod n.$$

Ker $p-1$ deli 10000!, po malem Fermatovem izreku (izrek 2.2) velja

$$m \equiv 1 \pmod{p} \text{ oz. } p \mid m-1.$$

Tako obstaja velika možnost, da n ne deli $m-1$ in zato je $g = \gcd(m-1, n)$ netrivialni faktor števila n. Za osnovo c lahko izberemo katerokoli število, ki je tuje številu n.

V praksi ne vemo, kako veliko je najmanjše praštevilo, zato lahko periodično preverjamo največjega skupnega delitelja $g = \gcd(c^{k!} - 1, n)$. V naši različici algoritma bomo največji skupni delitelj računali le na koncu **for** zanke. Če je $g = 1$ nadaljujemo, če pa je slučajno enak n, to pomeni, da smo naenkrat našli vse delitelje števila n. V tem primeru moramo izbrati drugo vrednost za c. Ko je g katerokoli drugo število, smo našli netrivialni faktor števila n.

Algoritem 4 Pollardov $p-1$ algoritem

Vhod: sestavljeno liho število n, iskalna meja B, osnova c
Izhod: netrivialen faktor števila n

1: $m = c$
2: **for** $2 \leq i \leq B$ **do**
3: $\quad m = m^i \mod n$
4: **end for**
5: $g = \gcd(m-1, n)$
6: **if** $1 < g < n$ **then**
7: $\quad$ **return** g
8: **else**
9: $\quad$ **return failure**
10: **end if**

Kakšna je časovna zahtevnost algoritma? V **for** zanki vsakič izvedemo potenciranje - uporabimo lahko algoritem kvadriraj in zmnoži[1], ki ima časovno zahtevnost $\mathcal{O}(\log B(\log n)^2)$. Na koncu zanke izračunamo še največji skupni delitelj. Če za njegov izračun uporabimo razširjen Evklidov algoritem (algoritem 1), porabimo $\mathcal{O}((\log n)^3)$ časa. Tako je skupna časovna zahtevnost Pollardovega $p-1$ algoritma $\mathcal{O}(B \cdot \log B(\log n)^2 + (\log n)^3)$.

Ideja Pollardovega $p-1$ algoritma je uporabljena tudi v Lenstrovi metodi (ECM), ki je opisana v poglavju 2.2.3.

2.2.2 Williamsov $p+1$ algoritem

Williamsov $p+1$ algoritem ima podobno idejo kakor Pollardov $p-1$ algoritem: primeren je za tista števila n, ki imajo vsaj en praštevilski faktor p, za katerega velja, da ima $p+1$ le majhne faktorje. Za potrebe našega algoritma bomo predpostavili, da $p+1$ deli 10000!. Algoritem uporablja Lucasovo zaporedje in je enakovreden Pollardovemu $p-1$ algoritmu.

Kvadratni ostanki in Legendrov simbol

Definicija 2.3 (Kvadratni ostanek po modulu p)**.** Naj bo p liho praštevilo in $a \in \mathbb{N}$. Število a je **kvadratni ostanek po modulu** p, če velja $a \not\equiv 0 \pmod{p}$ in ima kongruenca $y^2 \equiv a \pmod{p}$ rešitev $y \in \mathbb{Z}_p$.

Zanima nas, kako lahko določimo, ali je neko naravno število a kvadratni ostanek. Pri tem nam pomaga Eulerjev kriterij.

Izrek 2.4 (Eulerjev kriterij)**.** *Naj bo p liho praštevilo. Potem je a kvadratni ostanek po modulu p natanko tedaj, ko velja*

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

Eulerjev kriterij lahko učinkovito uporabimo za preverjanje, ali je neko število kvadratni ostanek. Če uporabimo algoritem kvadriraj in zmnoži, je časovna zahtevnost preverjanja $\mathcal{O}((\log p)^3)$.

Podajmo še definicijo Legendrovega simbola.

[1] Več o algoritmu lahko bralec najde v [St06] na strani 176.

Definicija 2.5 (Legendrov simbol)**.** Naj bo p liho praštevilo. Za naravno število a definirajmo **Legendrov simbol** $\left(\frac{a}{p}\right)$:

$$\left(\frac{a}{p}\right) = \begin{cases} 0, & \text{če je } a \equiv 0 \pmod{p} \\ 1, & \text{če je } a \text{ kvadratni ostanek po modulu } p \\ 1, & \text{če } a \text{ ni kvadratni ostanek po modulu } p \text{ in } a \not\equiv 0 \pmod{p} \end{cases}$$

Lucasova zaporedja

Lucasova zaporedja so določena zaporedja, ki ustrezajo rekurzivni enačbi

$$x_n = P \cdot x_{n-1} - Q \cdot x_{n-2}, \tag{2.1}$$

kjer sta P in Q določeni celi števili. Vsako drugo zaporedje, ki ustreza zvezi (2.1), lahko zapišemo kot linearno kombinacijo Lucasovih zaporedij $U_n(P,Q)$ in $V_n(P,Q)$. Nekateri znani primeri Lucasovih zaporedij so Fibonaccijeva števila, Pellova števila, ...

Za podana parametra P in Q so **Lucasova zaporedja 1. razreda** $U_n(P,Q)$ definirana z:

$$\begin{aligned} U_0(P,Q) &= 0 \\ U_1(P,Q) &= 1 \\ U_n(P,Q) &= P \cdot U_{n-1}(P,Q) - Q \cdot U_{n-2}(P,Q) \quad (\text{za } n > 1) \end{aligned}$$

Lucasova zaporedja 2. razreda V_n pa so definirana z:

$$\begin{aligned} V_0(P,Q) &= 2 \\ V_1(P,Q) &= P \\ V_n(P,Q) &= P \cdot V_{n-1}(P,Q) - Q \cdot V_{n-2}(P,Q) \quad (\text{za } n > 1) \end{aligned}$$

Ni težko pokazati, da za $n > 0$ veljata naslednji zvezi:

$$\begin{aligned} U_n &= \frac{P \cdot U_{n-1} + V_{n-1}}{2} \\ V_n &= \frac{(P^2 - 4Q)U_{n-1} + P \cdot V_{n-1}}{2} \end{aligned}$$

Pri Williamsovem $p+1$ algoritmu bomo uporabili tudi naslednje zveze:

$$\begin{aligned} V_{2i} &= V_i^2 - 2Q^i, \\ V_{2i+1} &= V_i \cdot V_{i+1} - P \cdot Q^i, \\ D &= P^2 - 4Q \end{aligned}$$

Williamsov $p+1$ algoritem

Izberimo celo število A, ki naj bo večje od 2 in določa naslednje Lucasovo zaporedje $V(A,1)$:

$$\begin{aligned} V_0 &= 2 \\ V_1 &= A \mod n \\ V_j &= (A \cdot V_{j-1} - V_{j-2}) \mod n \end{aligned}$$

Tedaj vsako liho praštevilo p deli $\gcd(n, V_n - 2)$, ko je M večkratnik razlike $p - \left(\frac{D}{p}\right)$, kjer je $D = A^2 - 4$. Želimo, da je $\left(\frac{D}{p}\right) = -1$, saj to pomeni, da D ni kvadratni ostanek po modulu p. Če generiramo Lucasovo zaporedje s številom D, ki ustreza zgornjim pogojem, potem p deli $U_{10000!}$ in tako je $\gcd(n, U_{10000!}) > 1$. Največji skupni delitelj bo manjši od n, če obstaja vsaj eno praštevilo q, ki deli n in velja, da $q - \left(\frac{D}{q}\right)$ ne deli $U_{10000!}$.

Lema 2.6. *Naj bosta $U_i(P)$ in $V_i(P)$ i-ta člena Lucasovih zaporedij s parametri P, $Q = 1$ in $D = P^2 - 4$. Potem veljata naslednji zvezi:*

$$\begin{aligned} U_{m \cdot k}(P) &= U_k(P) \cdot U_m(V_k(P)), \\ V_{m \cdot k}(P) &= V_m(V_k(P)). \end{aligned}$$

Iz leme (2.6) je očitno, da je veliko lažje izračunati $V_{10000!}$ kakor $U_{10000!}$. K sreči lahko z $V_{10000!}$ poiščemo praštevilske faktorje p števila n, za katere velja, da $p+1$ deli 10000!.

Lema 2.7. *Naj bo V_i Lucasovo zaporedje s parametri P, $Q = 1$ in $D = P^2 - 4$. Naj bo p praštevilo, za katerega je $\left(\frac{D}{p}\right) = -1$ in naj bo n naravno število. Potem velja:*

$$V_{m \cdot (p+1)} \equiv 2 \pmod{p}.$$

To pomeni: če $p+1 \mid 10000!$, potem $p \mid V_{10000!} - 2$.

Kako naj izberemo P, da bo $D = P^2 - 4$ zadoščal $\left(\frac{D}{p}\right) = -1$? Izkaže se, da ne poznamo nobenega kriterija za izbiro P in imamo približno 50% možnosti, da bo naključni P ustrezen. V praksi poskusimo s 3 različnimi vrednostmi za P. Če nobena ni ustrezna, potem je velika verjetnost, da n nima praštevilskega faktorja p, za katerega bi veljalo $p+1 \mid 10000!$ in v tem primeru raje izberemo drug algoritem za faktorizacijo.

Algoritem 5 Williamsov $(p+1)$ algoritem

Vhod: sestavljeno število n, mejo `max`, naključno število P
Izhod: faktor števila n

```
1:  count = 1                          ▷ count je za 1 večja od števila izračunanih V_i
2:  v = P
3:  while gcd(v − 2, n) = 1 and count ≤ max do
4:      for i = 1 to 10 do
5:          W = (P^2 − 2) mod n
6:          pretvori count v binarni zapis
7:                  ▷ (t je dolžina tega zapisa, b_i je vrednost posameznega bita)
8:          for k = t − 1 down to 1 do
9:              x = (v · w − P) mod n
10:             v = (v^2 − 2) mod n
11:             w = (w^2 − 2) mod n
12:             if b_k = 0 then
13:                 w = x
14:             else
15:                 v = x
16:             end if
17:             P = v
18:             count=count+1
19:         end for
20:     end for
21: end while
22: return gcd(v − 2, n)
```

Časovna zahtevnost algoritma je $\mathcal{O}(q)$, kjer je q največji praštevilski faktor števila $p+1$.

2.2.3 Lenstrova metoda (ECM)

Lenstra je metodo zasnoval na eliptičnih krivuljah. Metoda je uporabna za števila, za katera odpove algoritem poskusno deljenje (algoritem 3), pa do 25-30 mestnih števil n.

Eliptična krivulja

Eliptična krivulja je krivulja definirana s predpisom

$$y^2 = x^3 + ax + b, \tag{2.2}$$

kjer sta a in b izbrana tako, da velja

$$4a^3 + 27b^2 \neq 0. \tag{2.3}$$

To nam zagotovi, da ima kubična enačba

$$z = x^3 + ax + b \tag{2.4}$$

3 različne rešitve.

Enačba (2.2) je rešljiva takrat, ko je desna stran pozitivna in tedaj je

$$y = \pm\sqrt{x^3 + ax + b}.$$

Če ima enačba (2.4) 3 realne rešitve, potem ima graf funkcije (2.2) obliko, kot je prikazano na sliki 2.1.

Krivulja ima zanimivo lastnost: nenavpična premica jo vedno seka v 3 točkah. Tretje presečišče izračunamo s pomočjo leme 2.8.

Lema 2.8. *Naj bosta* (x_1, y_1) *in* (x_2, y_2) *točki na eliptični krivulji*

$$y^2 = x^3 + ax + b, \quad 4a^3 + 27b^2 \neq 0.$$

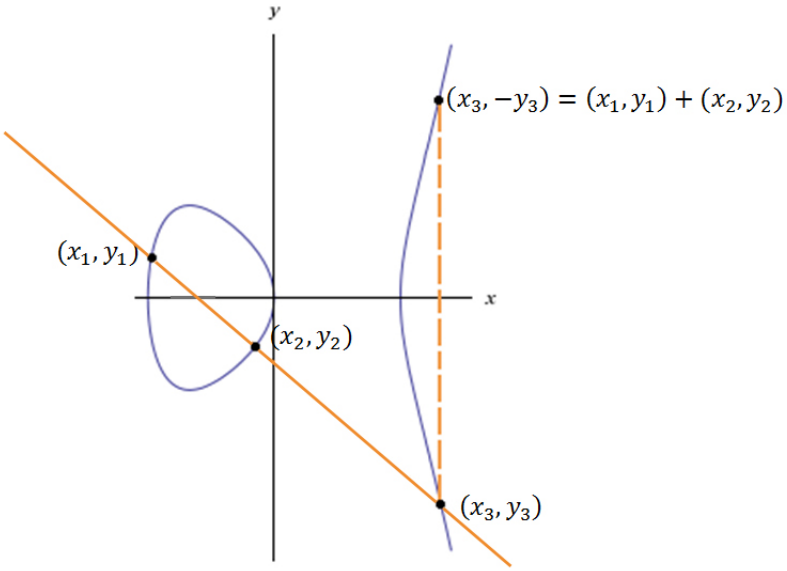
Ko je $x_1 = x_2$*, naj velja* $y_1 \neq y_2$*, vendar dovolimo, da je* $y_1 = y_2$ *ko velja* $y_1 \neq 0$*.*

Tretje presečišče (x_3, y_3) *izračunamo na naslednji način:*

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

Potem velja:

$$x_3 = \lambda^2 - x_1 - x_2$$
$$y_3 = \lambda(x_3 - x_1) + y_1$$



Slika 2.1: Primer eliptične krivulje.

Definicija 2.9. Za podano eliptično krivuljo in 2 racionalni točki na krivulji (x_1, y_1) in $(x_2, y_2) \neq (x_1, -y_1)$ definirajmo binarno operacijo ∂:

$$(x_1, y_1)\partial(x_2, y_2) = (x_3, -y_3),$$

kjer je točka (x_3, y_3) definirana kot v lemi 2.8.

Opazimo lahko, da vsota točk (x_1, y_1) in (x_2, y_2) ni točka (x_3, y_3), ampak njena preslikava čez abscisno os. Točka še vedno leži na eliptični krivulji.

Definirali smo množico racionalnih točk na eliptični krivulji in binarno operacijo ∂. Ker nas zanima definicija grupe na eliptični krivulji, si najprej oglejmo definicijo grupe.

Definicija 2.10 (Grupa)**.** Množica $\mathcal{G}$ opremljena z binarno operacijo $*$ je **grupa**, če velja:

- binarna operacija $*$ na $\mathcal{G}$ je **asociativna**:

$$(a * b) * c = a * (b * c)$$

- v množici $\mathcal{G}$ obstaja **nevtralni element** e za binarno operacijo $*$:

$$a * e = e * a = a$$

- za vsak element $a \in \mathcal{G}$ obstaja **inverzni element** a^{-1}, da velja:

$$a * a^{-1} = a^{-1} * a = e$$

Za definicijo grupe na eliptični krivulji potrebujemo še definicijo operacije $(x, y)\partial(x, -y)$, nevtralnega elementa in inverznega elementa.

Definicija 2.11. Definirajmo ∞ za nevtralni element binarne operacije ∂ in definirajmo:

$$(x, y)\partial(x, -y) = (x, -y)\partial(x, y) = \infty.$$

Točko ∞ lahko razumemo kot točko neskončno, skozi katero grejo vse navpične premice. S to definicijo smo tako določili tretje presečišče navpičnih premic z eliptično krivuljo in tako imamo vse potrebno za definicijo grupe na eliptični krivulji.

Definicija 2.12. Za dano eliptično krivuljo $y^2 = x^3 + ax + b,\ 4a^3 + 27b^2 \neq 0$ naj bo $E(a, b)$ grupa na množici racionalnih točk na krivulji skupaj s točko ∞, opremljena z binarno operacijo ∂.

Eliptične krivulje po modulu p

Najprej definirajmo operacijo ∂ po modulu n.

Definicija 2.13 (Eliptična krivulja po modulu p)**.** Naj bo ∞ nevtralni element in naj velja:

$$(x_1, y_1)\partial(x_2, y_2) = \infty,$$

ko je $x_1 \equiv x_2 \pmod{n}$ in $y_1 \equiv -y_2 \pmod{n}$. Kvocient λ izračunamo po naslednji enačbi:

$$\lambda = \begin{cases} (y_1 - y_2) \cdot ((x_1 - x_2)^{-1} \mod n) \mod n, \\ \qquad\qquad \text{ko je } x_1 \not\equiv x_2 \pmod{n} \text{ in } \gcd(x_1 - x_2, n) = 1 \\ (3x_1^2 + a) \cdot ((2y_1)^{-1} \mod n) \mod n, \\ \qquad\qquad \text{ko je } x_1 \equiv x_2 \pmod{n} \text{ in } \gcd(y_1 + y_2, n) = 1 \end{cases}$$

Sedaj lahko definiramo x_3 in y_3:

$$x_3 = (\lambda^2 - x_1 - x_2) \mod n,$$

$$y_3 = (\lambda(x_3 - x_1) + y_1) \mod n.$$

Binarna operacija ∂ po modulu n je podana z:

$$(x_1, y_1)\partial(x_2, y_2) \equiv (x_3, -y_3) \pmod{n},$$

ko sta x_3 in y_3 definirana. Izkaže se, da je binarna operacija definirana vedno, ko je n liho praštevilo.

Definicija 2.14 (Grupa na eliptični krivulji po modulu p)**.** Naj bo p liho praštevilo večje kot 3 in naj bosta a in b celi števili, za kateri velja

$$4a^3 + 27b^2 \not\equiv 0 \pmod{p}.$$

Potem je $E(a, b)/p$ grupa na eliptični krivulji po modulu p, katere elementi so pari (a, b) nenegativnih celih števil manjših kot p, ki zadoščajo

$$y^2 \equiv x^3 + ax + b \pmod{p},$$

s točko ∞ in binarno operacijo ∂ podano v definiciji 2.13

Lenstrova metoda (ECM)

Metodo, ki za faktorizacijo števil uporablja eliptične krivulje, sta razvila A. K. Lenstra in H. W. Lenstra, Jr.. Naj bo n sestavljeno število, ki je tuje številu 6. V praksi velja, da n nima majhnih faktorjev, algoritem pa poišče en faktor števila n.

1. Najprej izberemo eliptično krivuljo oblike $y^2 = x^3 + ax + b$ in točko $P = (x, y)$ na krivulji z naključnima neničelnima koordinatama. Nato izberemo še neničelni parameter $a \pmod{n}$ in izračunamo

   $$b = y^2 - x^3 - ax \pmod{n}.$$

2. Za točko P bomo izračunali produkt

   $$kP = \underbrace{P\partial P\partial \ldots \partial P}_{k}.$$

   Operacija ∂ je definirana v lemi 2.8.

   Pri izračunu računamo tudi kvocient $\lambda = \dfrac{u}{v}$. Ko velja $\gcd(u, n) = 1$ in je imenovalec $v = 0 \pmod{n}$, bo rezultat $P\partial Q = \infty$. Ko $\gcd(u, n)$ ni 1 ali n, potem vsota ne bo točka na krivulji, ampak nam bo $\gcd(v, n)$ dal netrivialni faktor števila n.

3. Izračunamo tudi eP na eliptični krivulji po modulu n, kjer je e produkt veliko majhnih števil, npr. $B!$. To lahko učinkovito izračunamo - v vsakem koraku dodamo en majhen faktor. Da izračunamo $B!P$ prvo izračunamo $2P$, nato $3(2P)$, $4(3!P)$, ... Seveda mora biti B dovolj majhen, da je operacija izvedljiva v realnem času.

4. - Če pri računanju ne naletimo na neobrnljive elemente po modulu n ali dobimo $kP = \infty$, moramo izbrati novo krivuljo in začetno točko,
   - če dobimo $\gcd(v, n) \neq 1$ ali n, potem smo našli netrivialni faktor števila n.

Algoritem 6 Lenstrova metoda (ECM)

Vhod: sestavljeno število n, za katerega velja $\gcd(n, 6) = 1$, meja B
Izhod: faktor števila n

izberi naključne $x, y, a \in [0, n-1]$
$b = (y^2 - x^3 - ax) \mod n$
$g = \gcd(4a^3 + 27b^2, n)$
if $g = n$ **then**
    Neuspešna izbira parametrov - pojdi na 1
else if $g > 1$ **then**
    return g
end if
$E =$ eliptična krivulja, ki je določena s parametroma a in b
$P = (x, y)$
for $1 \leq i \leq \pi(B)$ **do**
    poišči največje celo število a_i, da velja $p_i^{a_i} \leq B$
    for $1 \leq j \leq a_i$ **do**
        $P = p_i P$
        if $\gcd(v, n) \neq 1$ ali n **then**
            return $\gcd(v, n)$
        end if
    end for
end for
Neuspešna izbira parametrov - pojdi na 1

Podali bomo hevristično oceno časovne zahtevnosti algoritma[2].

[2]Podrobnejšo izpeljavo si bralec lahko prebere v [CP00] (poglavje 7.4.1).

Naj bo p najmanjši praštevilski faktor števila n. Hevristična ocena temelji na verjetnosti, da za nek k velja $kP = 0$. Izkaže se, da je ocena časovne zahtevnosti algoritma

$$\mathcal{O}\left(e^{(\sqrt{2}+\mathcal{O}(1))(\sqrt{\ln p \ln\ln p}}\right).$$

Kakor lahko opazimo, je časovna zahtevnost neodvisna od velikosti faktoriziranega števila n, zato je metoda primerna za števila, ki so po velikosti prevelika celo za kvadratno sito, a imajo majhne praštevilske faktorje.

Pri tem omenimo, da $\mathcal{O}(1)$ z naraščajočim p limitira proti 0. To pomeni, da je število korakov algoritma sorazmerno z velikostjo najmanjšega praštevilskega faktorja p. V najslabšem primeru, ko je n produkt dveh velikih praštevil, je časovna zahtevnost enaka časovni zahtevnosti algoritma kvadratno sito (poglavje 3.2). Vendar pa sta zaradi velike natančnosti posameznega koraka Lenstrovega algoritma za take primere bolj uporabna algoritma kvadratno sito ali številsko sito (poglavje 3.4). Če ne vemo, ali število n spada med najslabše primere, najprej uporabimo ECM in ga po določenem času, če nam ne vrne rezultata, zamenjamo s kvadratnim ali številskim sitom.

Kaj pa naredimo, ko je število preveliko za ta dva algoritma? V tem primeru nam ostane le uporaba Lenstrovega algoritma, ki nam uspešno faktorizira tudi veliko število n, če ima n dovolj majhen najmanjši praštevilski faktor ali ko je implementacija algoritma taka, da hitro naletimo na parametre, s katerimi algoritem uspešno faktorizira n.

2.3 Pollardov ρ algoritem

Ideja Pollardovega ρ algoritma je naslednja: naj bo p najmanjši praštevilski faktor števila n in naj obstajata dve števili $x, x' \in \mathbb{Z}_n$, da velja $x \neq x'$ in $x \equiv x' \pmod{p}$. V tem primeru velja $p \leq \gcd(x - x', n) < n$ in tako lahko izračunamo netrivialni faktor števila n.

Naj bo f polinom s celimi koeficienti, npr. $f(x) = x^2 + a$, kjer je a majhna konstanta (velikokrat je $a = 1$). Poglejmo zaporedje $x_1, x_2, \ldots$, kjer naj bo $x_1 \in \mathbb{Z}_n$, za $j \geq 2$ pa velja:

$$x_j = f(x_{j-1}) \pmod{n}. \tag{2.5}$$

Za $m \in \mathbb{N}$ definirajmo množico $X = \{x_1, x_2, \ldots, x_m\}$. Množico X si lahko predstavljamo kot množico različnih ostankov po modulu n. Želimo, da bi bila X naključna podmnožica množice $\mathbb{Z}_n$ moči n.

V množici X želimo poiskati dve različni vrednosti $x_i, x_j \in X$, da velja $\gcd(x_j - x_i, n) > 1$. Kako bi poiskali takšna elementa množice? Ena možnost

je, da bi za vsak element x_j, ki bi ga dodali množici X, izračunali $\gcd(x_j - x_i, n)$ z vsemi ostalimi elementi x_i množice X. To je preprosta in dobra ideja, vendar zahteva veliko izračunov največjega skupnega večkratnika, zato si oglejmo drugačno idejo.

Naj bo $x_i \equiv x_j \pmod{p}$. Ker ima polinom f celoštevilske koeficiente, velja $f(x_i) \equiv f(x_j) \pmod{p}$. Po zvezi (2.5) lahko zapišemo

$$x_{i+1} \mod p = (f(x_i) \mod n) \mod p = f(x_i) \mod p,$$

saj p deli n. Podobno:

$$x_{j+1} \mod p = f(x_j) \mod p.$$

Tako velja $x_{i+1} \equiv x_{j+1} \pmod{p}$. Z indukcijo lahko to zvezo posplošimo: če je $x_i \equiv x_j \pmod{p}$, potem velja $x_{i+\delta} \equiv x_{j+\delta} \pmod{p}$ za vsak $\delta \geq 0$.

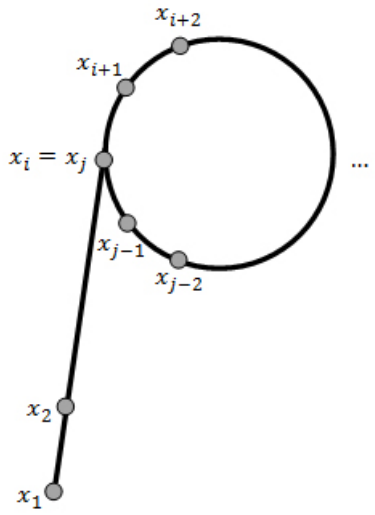
Hitro lahko opazimo, da ima naše zaporedje zanimivo obliko, ki je prikazana na sliki 2.2 in spominja na grško črko ro (ρ), po kateri je tudi poimenovan algoritem. Prvih i elementov zaporedja tvori njegov "rep":

$$x_1 \mod p \to x_2 \mod p \to \ldots \to x_i \mod p,$$

preostali elementi pa tvorijo cikel:

$$x_i \mod p \to x_{i+1} \mod p \to \ldots \to x_j \mod p = x_i \mod p.$$

Naš cilj je ugotoviti, kdaj velja $x_i \equiv x_j \pmod{p}$.



Slika 2.2: Zaporedje v obliki črke ro (ρ).

Algoritem 7 Pollardov ρ algoritem

Vhod: sestavljeno število n, vrednost x_1, ki je prva vrednost za x, in polinom $f(x)$
Izhod: netrivialni faktor števila n ali **failure**, ko iskanje ni bilo uspešno

1: $x = x_1$
2: $x' = f(x) \mod n$
3: $p = \gcd(x - x', n)$
4: **while** $p = 1$ **do** ▷ V i-ti iteraciji je $x = x_i$ in $x' = x_{2i}$
5: $x = f(x) \mod n$
6: $x' = f(x') \mod n$
7: $x' = f(x') \mod n$
8: $p = \gcd(x - x', n)$
9: **end while**
10: **if** $p = n$ **then**
11: **return failure**
12: **else**
13: **return** p
14: **end if**

Pričakovano število iteracij algoritma je $\sqrt{p}$ in ker velja $p < \sqrt{n}$, je pričakovana časovna zahtevnost algoritma $\mathcal{O}\left(\sqrt[4]{n}\right)$.

2.4 Frematov algoritem

Ideja Frematovega algoritma je, da število n razbijemo na produkt dveh naravnih števil a, b: $n = a \cdot b$. Vsako izmed teh dveh števil testiramo, ali je praštevilo, in če ni, na njem ponovno uporabimo algoritem.

Frematov algoritem se danes uporablja le tedaj, ko vemo, da ima n dva faktorja, ki sta blizu $\sqrt{n}$, vendar pa je sama ideja algoritma uporabljena v dveh zelo razširjenih algoritmih: kvadratno sito (poglavje 3.2) in v algoritmu za faktorizacijo z verižnimi ulomki (CFRAC) (poglavje 3.3.2).

Naj bo n podano število, ki ga želimo faktorizirati. Če ga zapišemo kot razliko dveh popolnih kvadratov:

$$n = x^2 - y^2,$$

potem velja

$$n = (x - y)(x + y).$$

Tako smo uspeli n zapisati kot produkt dveh manjših faktorjev.

Povsem varno je trditi, da je n liho število, saj v nasprotnem primeru poznamo njegov faktor. Naj bo $n = a \cdot b$, kjer sta a in b oba liha, kar lahko sklepamo iz lihosti števila n. Tedaj velja:

$$x = \frac{a+b}{2} \quad \text{in} \quad y = \frac{a-b}{2}.$$

Dokaz.

$$\begin{aligned} x^2 - y^2 &= \frac{1}{4}(a^2 + 2ab + b^2 - a^2 + 2ab - b^2) \\ &= a \cdot b \\ &= n \end{aligned}$$

□

Fermatov algoritem deluje v nasprotni smeri kakor algoritem poskusno deljenje (algoritem 3). Tam smo začeli pregledovati majhne faktorje in pregledali vse do $\sqrt{n}$, tukaj pa začnemo s faktorji blizu $\sqrt{n}$ in jih zmanjšujemo.

Prvo izberemo x, ki je enak najmanjšemu celemu številu, ki je večji ali enak $\sqrt{n}$. Število y je na začetku enako 0 in ga v vsaki ponovitvi povečamo, dokler ni razlika $x^2 - y^2$ enaka ali manjša od n. Ko je razlika enaka n, smo našli iskana faktorja, sicer pa povečamo x in poskusimo znova.

Algoritem 8 Fermatov algoritem

Vhod: število n
Izhod: iskani vrednosti a in b

1: $x = \lceil \sqrt{n} \rceil$
2: $y = x^2 - n$
3: **while** y ni kvadrat **do**
4: $\quad x = x + 1$
5: $\quad y = x^2 - n$
6: **end while**
7: $a = x - \sqrt{y}$
8: $b = x + \sqrt{y}$
9: **return** a, b

Algoritem v glavni **while** zanki ne vsebuje operacij množenja ali deljenja, zato se zanka izvaja zelo hitro. Večji problem je število ponovitev zanke, ki je lahko zelo veliko.

Ena izmed pohitritev algoritma je ugotovitev, da r do konca zanke ne bo enak 0, če $r = x^2 - n$ ni popoln kvadrat. Zato lahko pred zanko preverimo, ali je število popoln kvadrat ali ne, in si na tak način vsaj malo zmanjšamo število ponovitev zanke.

Kakšna pa je časovna zahtevnost algoritma? Če je eden izmed faktorjev a blizu $\sqrt{n}$, potem algoritem za izračun faktorjev potrebuje le nekaj korakov (ne glede na velikost števila n). V najslabšem primeru, ko je n praštevilo, pa algoritem potrebuje kar $\mathcal{O}(n)$ korakov, zato to metodo največkrat kombiniramo z algoritmom poskusno deljenje.

Kraitchikova izboljšava

Leta 1920 je Maurice Kraitchik predstavil zanimivo izboljšavo Fermatovega algoritma, ki je osnova večine modernih algoritmov za faktorizacijo naravnih števil.

Glavna ideja Kraitchikove izboljšave je, da namesto iskanja x in y, ki zadoščata zvezi

$$x^2 - y^2 = n,$$

poiščemo "naključna" x in y, za katera velja

$$x^2 \equiv y^2 \pmod{n}.$$

Par (x, y) nam ne zagotavlja faktorizacije, vendar pa zagotavlja, da n deli $x^2 - y^2 = (x - y)(x + y)$ in tako imamo 50% možnosti, da nam bo $\gcd(n, x - y)$ dal netrivialni faktor števila n.

Kraitchikova izboljšava se danes uporablja v dveh znanih algoritmih:

- Shanksova faktorizacija s kvadratnimi formami (SQUFOF, poglavje 3.3.3), ki za faktorizacijo uporablja verižne ulomke in Kraitchikovo idejo,
- algoritem za faktorizacijo z verižnimi ulomki (CFRAC, poglavje 3.3.2) pa temelji na verižnih ulomkih in ideji, ki sta jo razvila Lehmerja in Powersa.

2.5 Eulerjev algoritem

Glavna ideja Eulerjevega algoritma je, da najdemo faktorje števila n s pomočjo dveh različnih predstavitev števila n kot vsote dveh kvadratov:

$$n = a^2 + b^2 = c^2 + d^2. \tag{2.6}$$

Na primer:

$$221 = 10^2 + 11^2 = 5^2 + 14^2$$

Eulerjeva metoda deluje le za števila, ki jih lahko zapišemo kot vsoto dveh kvadratov: $n = a^2 + b^2$. Ker predvidevamo, da je n liho število, lahko brez škode za splošnost trdimo, da je a liho število in b sodo število.

Tudi pri drugi predstavitvi števila n ($n = c^2 + d^2$) naj velja, da je c liho število in d sodo število. Zvezo (2.6) lahko preoblikujemo:

$$\begin{aligned} a^2 + b^2 &= c^2 + d^2 \\ a^2 - c^2 &= d^2 - b^2 \\ (a-c)(a+c) &= (d-b)(d+b) \end{aligned} \tag{2.7}$$

Naj bo k največji skupni delitelj števil $(a-c)$ in $(d-b)$, tako da velja:

$$a - c = k \cdot l, \quad d - b = k \cdot o, \quad \gcd(l, o) = 1$$

Ker sta $(a-c)$ in $(d-b)$ lihi števili, je tudi k liho število. Ko izračunano vstavimo v zvezo (2.7), dobimo:

$$k \cdot l(a+c) = k \cdot o(d+b) \tag{2.8}$$

Ker sta l in o med seboj tuji števili, mora biti $(a-c)$ deljiv z o, zato lahko zapišemo:

$$(a-c) = r \cdot o. \tag{2.9}$$

Ko to uporabimo v (2.8), dobimo:

$$l \cdot r \cdot o = o(d+b)$$

$$d + b = l \cdot r \tag{2.10}$$

Iz zvez (2.9) in (2.10) ugotovimo, da je $\gcd(a+c, d+b) = r$, kar pomeni, da je tudi r liho število.

Število n lahko sedaj zapišemo kot produkt:

$$n = \left(\left(\frac{k}{2}\right)^2 + \left(\frac{r}{2}\right)^2 \right) \left(o^2 + l^2\right). \tag{2.11}$$

Dokaz. Preverimo, ali je zveza pravilna:

$$\begin{aligned} n &= \frac{1}{4}(k \cdot o)^2 + \frac{1}{4}(r \cdot o)^2 + \frac{1}{4}(k \cdot l)^2 + \frac{1}{4}(r \cdot l)^2 \\ &= \frac{1}{4}\left((d-b)^2 + (a+c)^2 + (a-c)^2 + (d+b)^2\right) \\ &= \frac{1}{4}\left(d^2 - 2db + b^2 + a^2 + 2ac + c^2 + a^2 - 2ac + c^2 + d^2 + 2db + b^2\right) \\ &= \frac{1}{4}\left(2d^2 + 2b^2 + 2a^2 + 2c^2\right) \\ &= \frac{1}{4}\left(2n + 2n\right) \\ &= n \end{aligned}$$

□

Očitno je, da zveza (2.11) nikoli nima trivialnih faktorjev.

Ostaja nam še vprašanje, kako na dva načina zapisati število n kot vsoto dveh kvadratov. Postopek je podoben kot pri Fermatovem algoritmu (poglavje 2.4): izračunamo razliko $n - x^2$ za različne x in preverimo, ali je dobljena razlika popolni kvadrat. Da zmanjšamo število potenciranj, lahko uporabimo naslednjo zvezo:

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

Primer. *Naj bo* $n = 2501$*. Prvo predstavitev najdemo takoj:*

$$2501 = 50^2 + 1,$$

drugo pa poiščemo s pomočjo zgornje zveze:

$$n - (x-1)^2 = 2501 - (50-1)^2 = 100 = 10^2$$

Tako smo našli obe predstavitvi števila 2501*:*

$$2501 = 50^2 + 1 = 49^2 + 10^2.$$

Sedaj lahko uporabimo prej opisan postopek:

$a = 1$	$a - c = -48$	$k = 8$
$b = 1$	$a + c = 50$	$l = -6$
$c = 1$	$d - b = -40$	$o = -5$
$d = 1$	$d + b = 60$	$r = 10$

in zapišemo n *kot produkt dveh števil:*

$$2501 = (4^2 + 5^2)(5^2 + 6^2) = 41 \cdot 61$$

Izkaže se, da lahko z Evklidovim algoritmom poiščemo vse možne faktorje danega števila, če to število ustreza pogojem metode.

Algoritem 9 Eulerjev algoritem

Vhod: število n
Izhod: faktor števila n oz. javi, da je število praštevilo

if n je potenca nekega praštevila p **then**
  return p
end if
$x_0 = \lfloor \sqrt{n - \sqrt[3]{n^2}} \rfloor$
$d = n - x_0^2$ ▷ $d \approx \sqrt[3]{n^2}$
if $\gcd(d, n) \neq n$ ali 1 **then**
  return $\gcd(d, n)$
end if
z nekim algoritmom za faktorizacijo (npr. algoritem poskusno deljenje) preveri faktorje števila n, ki so manjši kot $\sqrt[4]{\frac{4d}{3}}$ - če najdeš netrivialni faktor ga vrni
for $1 \leq a \leq \sqrt{\frac{4d}{3}}$ **do**
  if a je kvadrat po modulu d **then**
    poišči rešitev (x_1, y_1) enačbe $an = x^2 + dy^2$: $x_1, y_1 \in \mathbb{N}, y^2 \neq a$
  end if
end for
if nismo našli rešitve **then**
  return n je praštevilo
else ▷ Veljata zvezi: $n = x_0^2 + d$ in $an = x_1^2 + dy_1^2$
  return $g = \gcd(n, x_0 y_1 - x_1)$
end if

Algoritem

Algoritem, ki ga najdemo v [Mc96], je primeren za vsa števila in je posplošitev Eulerjeve metode. Zato si najprej oglejmo, kakšne so razlike.

Število n želimo zapisati kot $x^2 + dy^2$ na dva načina z istim koeficientom d:

$$n = x_1^2 + dy_1^2 = x_2^2 + dy_2^2.$$

Potem velja

$$(x_1 y_2)^2 \equiv (x_2 y_2)^2 \pmod{n}$$

in to pomeni, da bo največji skupni delitelj števil n in $x_1y_2 - x_2y_1$ netrivialni faktor števila n vedno, ko $x_1y_2 \equiv \pm x_2y_1 \pmod{n}$.

Časovna zahtevnost algoritma je $\mathcal{O}\left(n^{\frac{1}{3}+\varepsilon}\right)$.

Poglavje 3

Splošni algoritmi

Splošni algoritmi so tisti algoritmi, pri katerih je časovna zahtevnost odvisna od velikosti števila n. Primerni so za faktorizacijo večjih števil, pri katerih namenski algoritmi odpovejo. Med splošne algoritme uvrščamo:

- Dixonov algoritem,
- kvadratno sito (QS),
- faktorizacija z verižnimi ulomki (CFRAC),
- Shankova faktorizacija s kvadratnimi formami (SQUFOF) in
- številsko sito (NFS).

3.1 Dixonov algoritem

Dixonov algoritem je izboljšava Fermatovega algoritma (poglavje 2.4), ki poišče taki števili x in y, da velja:

$$x^2 \equiv y^2 \pmod{n}$$

in tako obstaja dovolj velika verjetnost, da bo $\gcd(n, x - y)$ dal netrivialen faktor števila n.

Izberemo naključno celo število r in izračunamo:

$$g(r) = r^2 \mod n.$$

Nato faktoriziramo $g(r)$ - ponavadi izberemo preprost algoritem, recimo algoritem poskusno deljenje (algoritem 3), in pregledamo števila do 10000. Če

faktorizacija $g(r)$ ne uspe, izberemo nov r. Postopek ponavljamo dokler nimamo več faktoriziranih $g(r)$, kakor je prašteval do meje, ki smo jo postavili za poskusno deljenje. V našem primeru, ko je meja 10000, bi potrebovali vsaj 1229 vrednosti za r, za katere lahko faktoriziramo $g(r)$, v splošnem pa bomo to vrednost označili s t.

Naj bo $p_1, p_2, \ldots, p_t$ prvih t praštevil. Če se da $g(r)$ popolnoma faktorizirati, ga lahko zapišemo kot

$$g(r) = p_1^{a_1} \times p_2^{a_2} \times \ldots \times p_t^{a_t},$$

kjer je večina a_i enaka 0. Faktorizacijo $g(r)$ lahko zapišemo in shranimo kot vektor:

$$\vec{v}(r) = (a_1, a_2, \ldots, a_t).$$

Če so vse komponente sode, potem je $g(r)$ popolni kvadrat in smo rešili naš problem, saj je $g(r) \equiv r^2 \pmod{n}$. Vendar pa se to zgodi le redko. V ostalih primerih si lahko pomagamo z velikim številom izračunanih vektorjev $\vec{v}(r)$. Najprej izračunajmo $w(r)$:

$$w(r) = (b_1, b_2, \ldots, b_t), \text{ kjer je } b_i = \begin{cases} 0, & \text{ko je } a_i \text{ sod} \\ 1, & \text{ko je } a_i \text{ lih} \end{cases}$$

Na njih uporabimo Gaussovo eliminacijo po modulu 2, ter tako poiščemo podmnožico $r$jev, za katere ima vsota njihovih $v(r)$ sode vse komponente vektorja. Produkt ujemajočih $g(r)$ pa je zato popolni kvadrat:

$$g(r_1) \times g(r_2) \times \ldots \times g(r_t) \equiv r_1^2 \times r_2^2 \times \ldots \times r_t^2 \pmod{n}.$$

Verjetnost, da tako najdemo faktor števila n, je približno 50%. Če nam ne uspe, poiščemo drugo podmnožico $r$jev, za katere so $w(r)$ linearno odvisni po modulu 2.

Metoda je verjetnostna, zato nimamo zagotovila, da bomo uspešno faktorizirali število n, vendar v praksi deluje zelo dobro in najde faktor hitreje kot deterministični algoritmi.

Časovna zahtevnost je odvisna od meje B. Če za B vzamemo vrednost $B = e^{\frac{1}{2}\sqrt{\ln n \ln(\ln n)}}$, je časovna zahtevnost algoritma $\mathcal{O}\left(e^{2+\mathcal{O}(1)\sqrt{\ln n \ln \ln n}}\right)$.

3.2 Kvadratno sito (QS)

Ideja, ki jo je predlagal Carl Pomerance leta 1981, temelji na tem, da bi v Dixonov algoritem vključili sito, podobno Eratostenovemu situ (algoritem 2).

Algoritem 10 Dixonov algoritem

Vhod: sestavljeno število n
Izhod: faktor števila n

for $1 \leq i \leq t+1$ **do**
  izberi r_i in izračunaj $g(r_i) \equiv r_i^2 \mod n$
  if $g(r_i)$ je B-gladko število **then**
    $g(r_i) = \prod_{j=1}^{t} p_j^{a_{ij}}$, kjer so p_j praštevila manjša od B
    Shrani par $(r_i, g(r_i))$
    $i = i + 1$
  end if
end for ▷ Shranjenih imamo $t+1$ parov $(r_i, g(r_i))$
Poišči β_j, da velja: $\sum_{j=1}^{t+1} \beta_j a_{ij}$ je sod za vsak i
$x = \prod_{j=1}^{t+1} r_j^{\beta_j}$
$y = \sqrt{\prod_{j=1}^{t+1} g(r_j)^{\beta_j}}$
return $\gcd(n, x - y)$

Kvadratno sito je na velikih številih prvi preizkusil Joseph Gerver, ki je uspešno faktoriziral 47-mestno število $3^{225}-1$. Leta 1994 pa je v osmih mesecih skupina znanstvenikov uspela faktorizirati 129-mestno število izziva RSA-129.

Pri Dixonovemu algoritmu izberemo naključni r, pri kvadratnem situ pa izračunamo $k = \lfloor \sqrt{n} \rfloor$ in nato za r izberemo $k+1, k+2, \ldots$, za $f(r)$ pa izberemo $f(r) = r^2 - n$, potem je $f(r) = g(r)$.

Poiskati želimo tiste $f(r)$, ki se faktorizirajo v praštevila, ki so manjša od 10000. Naj bo p liho praštevilo, ki je manjše od 10000. Predpostavimo, da smo že uporabili poskusno deljenje za n do 10000 in tako vemo, da p ne deli n. Če p deli $f(r)$, potem velja

$$n \equiv r^2 \pmod{p}$$

in je Legendrov simbol $\left(\frac{n}{p}\right) = 1$. Zato moramo upoštevati le tista praštevila, ki so manjša od 10000 in zanje velja $\left(\frac{n}{p}\right) = 1$, in tako moramo pregledati le polovico vseh praštevil. Množico praštevil, ki jih preizkušamo, ali delijo $f(r)$,

imenujemo **baza faktorjev**.

Algoritem kvadratno sito ima 3 korake:

1. Poišči bazo faktorjev in reši kongruenco $x^2 \equiv n \pmod{p}$ za vsako praštevilo p v bazi faktorjev.
2. S presejanjem poišči ustrezne $f(r)$, ki jih lahko popolnoma faktoriziramo v bazi faktorjev.
3. Z Gaussovo eliminacijo poišči produkt $f(r)$-jev, ki je popoln kvadrat.

Reševanje kvadratnih kongruenc

Prvo vprašanje, ki se nam pojavi, je kako velika naj bo baza faktorjev. Upoštevati moramo dva vidika: baza mora biti dovolj velika, da obstaja dovolj velika verjetnost, da bomo lahko faktorizirali dani $f(r)$, po drugi strani pa baza ne sme biti prevelika, saj delamo Gaussovo eliminacijo na matriki velikosti baze faktorjev.

Metoda je primerna za uporabo večkratnikov števila n - n lahko zamenjamo s številom $3n$ in tako povečamo verjetnost, da nam bo uspelo faktorizirati razliko $f(r) = r^2 - n$. Pri tem velja, da če praštevilo p deli $3n$, potem p deli $(r^2 - n)$ natanko tedaj, ko p deli r. Pomembno je, da večkratnik izberemo preden določimo bazo faktorjev.

Za vsako liho praštevilo p_i v bazi faktorjev moramo razrešiti kongruenco

$$x^2 \equiv n \pmod{p_i}, \tag{3.1}$$

kjer je n kvadratni ostanek po modulu p_i. V primerih, ko je $p_i \equiv 3 \pmod 4$ ali $p_i \equiv 5 \pmod 8$, lahko uporabimo Izrek 3.1. Izrek 3.2 velja za vsa liha praštevila, vendar je rahlo počasnejši od Izreka 3.1.

Izrek 3.1. *Če je n kvadratni ostanek po praštevilskem modulu p, je rešitev kongruence* (3.1)

$$x \equiv \begin{cases} n^{k+1} \mod p, & \text{ko } p = 4k+3 \\ n^{k+1} \mod p, & \text{ko } p = 8k+5 \text{ in } n^{2k+1} \equiv 1 \pmod p \\ (4n)^{k+1}\left(\frac{p+1}{2}\right) \mod p, & \text{ko } p = 8k+5 \text{ in } n^{2k+1} \equiv -1 \pmod p \end{cases}$$

Izrek 3.2. *Če je n kvadratni ostanek po praštevilskem modulu p in naj bo h izbran tako, da velja $\left(\frac{h^2-4n}{p}\right) = -1$. Z rekurzijo določimo zaporedje:*

$$v_1 = h, \quad v_2 = h^2 - 2n, \quad v_i = h \cdot v_{i-1} - n \cdot v_{i-2}.$$

Potem velja:

$$v_{2i} = v_i^2 - 2n^i \quad in \quad v_{2i+1} = v_i \cdot v_{i+1} - h \cdot n^i.$$

Tedaj je rešitev kongruence (3.1)*:*

$$x \equiv v_{\frac{p+1}{2}} \cdot \left(\frac{p+1}{2}\right) \pmod{p}$$

Kako izbrati ustrezni h? Izkaže se, da ga lahko izberemo popolnoma naključno, saj ima vsaka vrednost 50% možnosti, da ustreza pogoju.

Presejanje

Kako deluje sito je najlažje prikazati na primeru.

Primer. *Naj bo* $n = 4\ 999\ 486\ 012\ 441$*. V bazi faktorjev bomo imeli 30 praštevil, začetnih vrednosti za* r *pa naj bo* 10000*.*

Ker je $\lfloor\sqrt{n}\rfloor = 2235953$*, so števila* r *na intervalu* $(2230953, 2240954)$*:*

$$r = 2230953 + i; \quad 1 \leq i \leq 10000.$$

Razlika $(r^2 - n)$ *je zato blizu* 0 *in manjši kot je* $f(r)$*, boljše so naše možnosti, da bomo uspešno faktorizirali razliko. Ker dobimo tudi negativne razlike, moramo v bazi faktorjev imeti tudi število* -1*.*

Za vsako praštevilo p_i v bazi faktorjev mora biti n kvadratni ostanek po modulu p_i.

Primer (nadaljevanje)**.** *Ker je* n *kongruenten* 1 *po modulu* 8*, lahko* 8 *dodamo v bazo faktorjev, saj deli* $(r^2 - n)$ *vedno, ko je* r *lih. Iz seznama praštevil izberemo* 28 *praštevil, ki zadoščajo pogoju*

$$\left(\frac{n}{p}\right) = 1.$$

V našem primeru dobimo naslednja praštevila:

3	19	59	163	229	277	359
5	31	61	181	241	311	267
7	43	67	193	263	331	389
17	47	107	197	271	349	397

Za vsako praštevilo iz baze faktorjev moramo izračunati

$$n \equiv t^2 \pmod{p}_i.$$

Izberemo najmanjšo rešitev za t za ujemajoče praštevilo. Druga rešitev bi bila $p - t$.

Primer (nadaljevanje)**.** *Dobimo naslednjo rešitev:*

1	1	14	38	7	39	171
1	5	16	19	18	39	125
2	19	6	86	101	65	69
3	18	32	14	22	52	50

Sedaj imamo vse potrebne podatke za uporabo sita, zato si oglejmo Silvermanovo izboljšavo originalne metode.

Začnimo z vektorjem samih ničel, ki mu prištejemo $\log p_i$, ko p_i deli ujemajoč $f(r)$. Ko uporabimo sito na $2M$ vrednostih, je logaritem absolutne vrednosti $|(\lfloor\sqrt{n}\rfloor - M + i)^2 - n|$ približno

$$o = \frac{\log n}{2} + \log M.$$

Po presejanju dobimo dovolj majhno število vrednosti blizu o, da lahko z algoritmom poskusno deljenje na bazi faktorjev ugotovimo, katera vrednost se popolnoma faktorizira.

Bralca gotovo zanima, kaj pomeni dovolj blizu? Če je preostali nefaktoriziran del manjši kot koren največjega praštevila v bazi faktorjev, potem je praštevilo. Čeprav nimamo popolne faktorizacije v bazi faktorjev, vseeno dobimo uporabno popolno faktorizacijo. Silverman je predlagal, da določimo

$$c = o - T \cdot \log p_{\max},$$

kjer je $p_{\max}$ največje praštevilo v bazi faktorjev in T konstanta blizu števila 2. Za 13-mestna števila je $T = 1,5$ primerna izbira.

Primer (nadaljevanje)**.** *V našem primeru je za sode vrednosti i r lih, zato v prvem prehodu prištejemo* $\log 8$ *vsakemu drugemu elementu. Ker je* 2230953 *deljiv s* 3 *prištejemo* $\log 3$ *vnosom z indeksi* $1 + 3k$ *in* $2 + 3k$*,* $k = 0, 1, 2 \ldots$.

V splošnem velja

$$r = \lfloor\sqrt{n}\rfloor - M + i,$$

če presejamo na intervalu dolžine $2M$. Pri podanem lihem praštevilu p in rešitvi t kongruence (3.1) (t med 0 in p), je prva vrednost za i, ki ji prištejemo $\log p$, enaka

$$p + (t - (\lfloor\sqrt{n}\rfloor - M) \mod p).$$

Pri tem ne pozabimo na naslednjo lastnost: če p ne deli večkratnika števila n, potem imamo dve rešitvi kongruence (3.1): t in $p - t$.

Gaussova eliminacija

Gaussovo eliminacijo uporabimo za iskanje produkta vrednosti $f(r)$, ki tvorijo popolni kvadrat. Pri tem bomo uporabili Wiedemannovo izboljšavo, ki deluje na $GF(2)$. $GF(2)$ je Galaisov obseg, ki je izomorfen $\mathbb{Z}_2$. Za vsak i, za katerega lahko faktoriziramo $f(2230953 + i)$, tvorimo binarni vektor dolžine 30 - vsaka njegova komponenta ustreza enemu izmed 30 praštevil v bazi faktorjev. Vrednost i-te komponente je 1, če je v faktoriziranem zapisu ujemajoče praštevilo p_i lihe stopnje, in 0, če je sode. Bazni faktor -1 zapišemo kot zadnjo komponento vektorja.

Primer (nadaljevanje)**.** 39 *faktoriziranih števil tvori matriko dimenzije* 39×30. *Da lahko ugotovimo, katera kombinacija tvori popolni kvadrat, tej matriki na desni pripišemo enotsko matriko dimenzije* 39×39.

$$\begin{array}{l|l}
000000010010000010000000110001 & 100000000\ldots0 \\
000000000010000000001011010101 & 010000000\ldots0 \\
001000000000100000010010000101 & 001000000\ldots0 \\
000010000000000000010000011011011 & 000100000\ldots0 \\
000000100100000000000111010001 & 000010000\ldots0 \\
\vdots & \vdots
\end{array}$$

Postopek eliminacije začnemo s 30. stolpcem matrike. Poiščemo prvo vrstico, ki ima v tem stolpcu vrednost 1 *in jo po modulu* 2 *prištejemo vsem ostalim, ki imajo v tem stolpcu vrednost* 1. *Tako dosežemo, da ima vseh preostalih* 38 *vrstic v zadnjem stolpcu le vrednost 0. Ne pozabimo, da moramo isto vrstico prišteti tudi v enotski matriki.*

Postopek nadaljujemo v 29. stolpcu, kjer prav tako poiščemo prvo vrstico, ki ima vrednost 1, *ter jo prištejemo vsem ostalim z enako vrednostjo v tem stolpcu. To nadaljujemo dokler ne dobimo vrstice, ki ima na vseh* 30 *mestih vrednost* 0. *Zadnjih* 39 *števk (enotska matrika) pa nam pove, katere vrstice tvorijo popoln kvadrat.*

V našem primeru je prva vrstica, ki ima na prvih 30 *mestih same* 0*:*

$$0\ldots0000 \;\Big|\; 000000000000100010001000000000000100000$$

V enotski matriki je števka 1 *na* 13., 17., 21. *in* 34. *mestu, kar nam pove, da je produkt* $f(2230953+i)$ *za ujemajoče vrednosti* i *popoln kvadrat. Te vrednosti za* i *so:* 4340, 4786, 5083 *in* 8726*. Če zmnožimo te praštevilske vsote, dobimo:*

$$(2230953+4340)^2 \cdot (2230953+4786)^2 \cdot (2230953+5038)^2 \cdot (2230953+8726)^2 \equiv$$
$$\equiv 2^{16}\cdot 3^6 \cdot 5^4 \cdot 7^2 \cdot 17^2 \cdot 43^2 \cdot 193^2 \cdot 197^2 \cdot 241^2 \cdot 349^2 \pmod{4999486012441}$$

Tako smo dobili kongruenco oblike

$$x^2 \equiv y^2 \pmod{n}.$$

Sedaj lahko izračunamo x *in* y *po modulu* n*. V našem primeru je kvadratni koren po danem modulu enak za obe strani:* 2197001218533*. Verjetnost, da se zgodi tak primer, tj. da velja:*

$$\gcd(x-y,n) = 1 \text{ } ali \text{ } n,$$

je približno 50%*.*

Vendar pa ta neuspeh ne pomeni, da algoritem ne more faktorizirati števila n*, saj lahko poiščemo naslednjo vrstico, ki ima na prvih* 30 *mestih same* 0*. To je vrstica:*

$$0\ldots0000 \;\Big|\; 101010001111010010100010010100000000000$$

Ponovimo postopek in sedaj za x *in* y *dobimo:*

$$x = 2999903916061 \text{ } in \text{ } y = 1999582096380.$$

Vendar tudi tokrat nimamo sreče, saj je $\gcd(x-y,n) = 1$*. Šele v tretjem poskusu nam uspe:*

$$0\ldots0000 \;\Big|\; 101000010100001110010000111000110000000$$

V tem primeru za x *in* y *dobimo:*

$$x = 3665300235664 \text{ } in \text{ } y = 915847468484.$$

Največji skupni delitelj je

$$\gcd(x-y,n) = 999961$$

in tako smo uspešno faktorizirali število n*:*

$$4999486012441 = 999961 \cdot 4999681.$$

Časovna zahtevnost Gaussove eliminacije je približno

$$\mathcal{O}\left(B(w + B \cdot \ln(B) \cdot \ln\ln(B)\right),$$

kjer je B velikost baze faktorjev in w približno število operacij za množenje matrike z vektorjem. Ker je večina elementov v matriki enaka 0, je w zelo majhen. Za ta korak kvadratnega sita potrebujemo $2B^2$ prostora.

Časovna zahtevnost

Časovna zahtevnost algoritma kvadratno sito je odvisna od velikosti baze faktorjev, zato nas zanima, kakšna je njena optimalna velikost. Če izberemo majhno bazo faktorjev, bi potrebovali malo faktorizacij $f(r)$, da bi dobili verjetno faktorizacijo števila n. Vendar pa bi bilo iskanje popolne faktorizacije v majhni bazi zelo dolgotrajno, saj nima veliko števil vse faktorje iz majhne množice praštevil.

Če bi zgradili veliko bazo faktorjev in bi se tako skoraj vsa števila dalo faktorizirati v bazi faktorjev, pa bi nastal problem pri velikosti matrike, ki jo sestavimo za Gaussovo eliminacijo.

Zato je velikost baze faktorjev zelo pomembna - izkaže se, da je njena optimalna velikost približno

$$B = \left(e^{\sqrt{\ln(n)\cdot\ln(\ln(n))}}\right)^{\frac{\sqrt{2}}{4}}.$$

Interval, na katerem uporabimo sito, pa mora biti kub velikosti baze faktorjev:

$$M = \left(e^{\sqrt{\ln(n)\cdot\ln(\ln(n))}}\right)^{\frac{3\sqrt{2}}{4}}.$$

Nadaljnje analize pokažejo, da je uporaba sita trikratnik časa, ki ga porabimo za Gaussovo eliminacijo. Če sestavimo vse ocene, dobimo asimptotično časovno zahtevnost kvadratnega sita, ki je enaka:

$$\mathcal{O}\left(e^{\sqrt{1{,}125\cdot\ln(n)\cdot\ln\ln(n)}}\right).$$

Če uporabimo Wiedemannovo izboljšavo, se časovna zahtevnost zmanjša na:

$$\mathcal{O}\left(e^{\sqrt{\ln(n)\cdot\ln\ln(n)}}\right).$$

Algoritem 11 Kvadratno sito

Vhod: naravno število n in mejo za velikost praštevil v bazi faktorjev B
Izhod: netrivialni faktor števila n

1: $p_1 = 2$
2: $a_1 = 1$
3: poišči $k-1$ lihih praštevil

$$\{p_2, p_3, \dots p_k : p_i \leq B \text{ in } \left(\frac{n}{p}\right) = 1 \text{ za } i = 2, 3 \dots k\}$$

4: **for** $2 \leq i \leq k$ **do** ▷ Reševanje kvadratnih kongruenc
5: poišči $\pm x_i$, da velja $x_i^2 \equiv n \pmod{p_i}$
6: **end for**
7: **for** $0 \leq i \leq k+1$ **do** ▷ Presejanje
8: $r = \sqrt{n} + i$
9: v množico $\mathcal{S}$ dodaj $(r, r^2 - n)$
10: **end for**
11: **for** $(r, r^2 - n) \in \mathcal{S}$ **do**
12: izračunaj $r^2 - n = \prod_{i=1}^{k} p_i^{e_i}$
13: $\vec{v}(r^2 - n) = (e_1, e_2, \dots e_k)$
14: **end for**
15: Sestavi matriko dimenzije $(k+1) \times k$, kjer so vrstice vektorji $\vec{v}(x^2 - n)$, reducirani po modulu 2.
16: Z algoritmom linearne algebre poišči netrivialno podmnožico vrstic matrike, ki se seštejejo v ničelni vektor po modulu 2: $\{x'_1, x'_2, \dots x'_j\}$.
17: $x = x'_1 x'_2 \dots x'_j \mod n$
18: $y = \sqrt{(x'^2_1 - n)(x'^2_2 - n) \dots (x'^2_j - n)} \mod n$
19: $d = \gcd(x - y, n)$
20: **return** d

3.2.1 Kvadratno sito na družini polinomov (MPQS)

Kvadratno sito na družini polinomov je različica kvadratnega sita, ki namesto enega polinoma $(x^2 - n)$ uporabi družino polinomov. Njena glavna prednost je zmanjšanje intervala, na katerem presejamo in zato se poveča verjetnost, da bomo lahko faktorizirali polinome $f(x)$, ki so oblike:

$$f(x) = ax^2 + 2bx + c.$$

Prvo izberemo koeficient a, ki naj bo kvadrat naravnega števila, nato izberemo koeficient b, za katerega naj velja: $0 \leq b < a$ in $b^2 \equiv n \pmod{a}$, kar velja le, ko je n kvadrat po modulu q_i za vsako praštevilo q_i, ki deli a. Zato je pomembno, da izberemo tak a, za katerega poznamo faktorizacijo $a = \prod_i q_i^{e_i}$ in za vsak q_i velja $\left(\frac{n}{q}\right) = 1$. Nazadnje izberemo tak koeficient c, da velja $b^2 - 4ac = n$. Ko najdemo tak $f(x)$, opazimo, da

$$a \cdot f(x) = (ax)^2 + 2abx + ac = (ax + b)^2 - n$$

in tako velja

$$(ax + b)^2 \equiv a \cdot f(x) \pmod{n}.$$

Ker je a kvadrat, mora biti tudi $f(x)$ kvadrat.

Glavna prednost metode je zmanjšanje velikosti baze faktorjev in intervala presejanja: raziskovalci so ocenili, da se velikost baze faktorjev v primerjavi z originalnim kvadratnim sitom zmanjša na desetino, velikost intervala presejanja pa na eno tisočinko. Druga prednost metode je v paralelnem izvajanju, saj lahko vsak procesor računa z drugim polinomom in tako lahko računajo povsem neodvisno ter z osrednjim strežnikom komunicirajo le, ko presejajo na celem intervalu.

3.3 Faktorizacija z verižnimi ulomki

3.3.1 Verižni ulomki

Verižni ulomki so ulomki oblike

$$\frac{a_0}{a_1} = m_1 + \cfrac{1}{m_2 + \cfrac{1}{\ldots + \cfrac{1}{m_{n-1} + \cfrac{1}{m_n}}}},$$

pri čemer števila m_i $(i = 1, 2, \ldots n)$ izračunamo s pomočjo Evklidovega algoritma na naslednji način:

$$\begin{aligned}
&\frac{a_0}{a_1} = m_1 + \frac{a_2}{a_1}, \text{ kjer je } 0 < \frac{a_2}{a_1} < 1 \\
&\frac{a_1}{a_2} = m_2 + \frac{a_3}{a_2}, \text{ kjer je } 0 < \frac{a_3}{a_2} < 1 \\
&\ldots \\
&\frac{a_{n-2}}{a_{n-1}} = m_{n-1} + \frac{a_n}{a_{n-1}}, \text{ kjer je } 0 < \frac{a_n}{a_{n-1}} < 1
\end{aligned}$$

Števila a_i $(i = 1, 2 \ldots n)$ so naravna števila.

S pomočjo verižnih ulomkov lahko predstavimo racionalna ali iracionalna števila. Verižni ulomki racionalnih števil so končni, za iracionalna števila pa so neskončni, saj iracionalnih števil ne moremo zapisati s končno mnogo ulomki. Zato so verižni ulomki primerni za zapis kvadratnega korena števila n, ko n ni popolni kvadrat in potrebujemo aproksimacijo njegovega korena.

Če želimo poiskati aproksimacijo za $\sqrt{n}$, moramo poiskati celi števili x in y, ki zadoščata enačbi

$$x^2 - n \cdot y^2 = \pm 1. \tag{3.2}$$

Ko najdemo tak par, jih lahko s pomočjo izreka 3.3 generiramo neskončno mnogo.

Izrek 3.3. *Naj bo n naravno število, ki ni popoln kvadrat in za katerega ima enačba* (3.2) *vsaj eno rešitev a, b v naravnih številih. Potem ima enačba* (3.2) *neskončno mnogo rešitev, ki jih lahko rekurzivno izračunamo:*

$$\begin{aligned}
x_1 &= a \qquad & x_{i+1} &= a \cdot x_i + n \cdot b \cdot y_i \\
y_1 &= b \qquad & y_{i+1} &= b \cdot x_i + a \cdot y_i.
\end{aligned}$$

Bháskara-Brounckerjev algoritem

Bháskara-Brounckerjev algoritem izračuna aproksimacijo za $\sqrt{n}$, ki je enaka $\frac{P_i}{Q_i}$. Glavna ideja algoritma je, da večkrat uporabimo Evklidov algoritem (algoritem 1) - ko iščemo aproksimacijo za $\sqrt{n}$, uporabimo Evklidov algoritem na paru $\sqrt{n}$ in 1. Na prvi pogled je to nesmiselno početje, saj ne iščemo največjega skupnega delitelja teh dveh števil. Vendar pa je ideja večkrat ponoviti Evklidov algoritem - natančneje, ponavljamo ga, dokler v k-ti iteraciji ostanek ni

dovolj blizu števila 0 in tedaj ustavimo algoritem. Tako dobimo dovolj dobro aproksimacijo podanega korena.

Algoritem 12 Bháscara-Brounckerjev algoritem

Vhod: naravno število n
Izhod: vrednosti P_i in Q_i $\left(\sqrt{n} \approx \frac{P_i}{Q_i}\right)$ ter indeks i

1: $s = \lfloor \sqrt{n} \rfloor$
2: $B_0 = 0$
3: $B_1 = s$
4: $C_0 = 1$
5: $C_1 = n - s^2$
6: $P_0 = 1$
7: $P_1 = s$
8: $Q_0 = 0$
9: $Q_1 = 1$
10: $i = 1$
11: **while** $C_i \neq 1$ **do** ▷ Ko je $C_i = 1$ dobimo prvo rešitev enačbe (3.2)
12: $k = i - 1$
13: $j = i$
14: $i = i + 1$
15: $A_i = \lfloor \frac{s+B_j}{C_j} \rfloor$
16: $B_i = A_i \cdot C_j - B_j$
17: $C_i = C_k + A_i \cdot (B_j - B_i)$
18: $P_i = P_k + A_i \cdot P_j$
19: $Q_i = Q_k + A_i \cdot Q_j$
20: **end while**
21: **return** P_i, Q_i, i

Enačba

$$x^2 - n \cdot y^2 = 1 \tag{3.3}$$

bo imela rešitev v naravnih številih vedno, ko je n pozitiven in ni popolni kvadrat. Poiskati želimo prvo rešitev enačbe (3.3) in pri tem nam bosta v pomoč izreka 3.4 in 3.5.

Izrek 3.4. *Če sta a in b naravni števili, ki zadoščata enačbi*

$$a^2 - n \cdot b^2 = \pm 1,$$

potem je

$$\left|\frac{a}{b} - \sqrt{n}\right| = \frac{1}{b \cdot (a + b \cdot \sqrt{n})}.$$

Izrek 3.5. *Če za začetni vrednosti za izrek 3.3 vzamemo pozitivni rešitvi izreka 3.4, ki minimizirata numerično vrednost* $a + b\sqrt{n}$*, potem algoritem 12 generira vse pozitivne rešitve enačbe* (3.2)*.*

Vedno velja zveza:

$$P_i^2 - n \cdot Q_i^2 = (-1)^i \cdot C_i. \tag{3.4}$$

Primer. *V tabeli 3.1 so izračunane vrednosti za* $n = 13$*.*
Prvi rešitvi enačbe (3.2) *sta* 18 *in* 5*:*

$$18^2 - 13 \cdot 5^2 = -1,$$

naslednji pa 649 *in* 180*:*

$$649^2 - 13 \cdot 180^2 = 1.$$

i	A_i	B_i	C_i	P_i	Q_i
1	3	3	4	3	1
2	1	1	3	4	1
3	1	2	3	7	2
4	1	1	4	11	3
5	1	3	1	18	5
6	6	3	4	119	33
7	1	1	3	137	38
8	1	2	3	256	71
9	1	1	4	393	109
10	1	3	1	649	180
11	6	3	4	4287	1189

Tabela 3.1: Rešitev za $n = 13$.

3.3.2 Algoritem za faktorizacijo z verižnimi ulomki (CFRAC)

Osnovna ideja algoritma je enaka kot pri algoritmu kvadratno sito (poglavje 3.2): poiskati želimo rešitve kongruence

$$x^2 \equiv y^2 \pmod{n}. \tag{3.5}$$

Razlika med algoritmom za faktorizacijo z verižnimi ulomki in kvadratnim sitom je v tem, kako generiramo števila r. Pri metodi kvadratnega sita preprosto vzamemo števila, ki so najbližja $\sqrt{n}$. Razlika $r^2 - n$, ki jo moramo faktorizirati, pa je precej velika. Tudi če uporabimo različico MPQS z optimalnimi parametri, moramo faktorizirati število velikosti $\frac{M}{\sqrt{2}}\sqrt{n}$ in zato je verjetnost, da ga bomo uspeli faktorizirati s praštevili iz baze faktorjev, zelo majhna.

CFRAC izbere vrednosti r tako, da je njena absolutna vrednost manjša od $2\sqrt{n}$. Verjetnost, da jo bomo lahko faktorizirali v bazi, je zato večja. Vendar pa moramo za to verjetnost plačati ceno, saj je najboljši način za faktorizacijo $(r^2 \mod n)$ v bazi faktorjev algoritem poskusno deljenje (algoritem 3).

Algoritem za faktorizacijo z verižnimi ulomki za r uporabi števila P_i, ki jih generira algoritem 12. Iz enačbe (3.4) sledi:

$$P_i^2 \equiv (-1)^i \cdot C_i \pmod{n}$$

Algoritem 12 tako priredimo, da računamo le vrednosti P_i, saj Q_i ne potrebujemo. Tudi pri CFRAC dodamo v bazo faktorjev vrednost -1.

Z algoritmom poskusno deljenje poskušamo v bazi faktorjev faktorizirati vsak novi C_i. Če nam uspe, shranimo $P_i \mod n$, C_i in faktorizacijo za $(-1)^i \cdot C_i$, sicer pa z algoritmom 12 generiramo nov C_i. Ko število popolnoma faktoriziranih C_i preseže velikost baze faktorjev, uporabimo Gaussovo eliminacijo in z njo poiščemo produkt C_i-jev, ki tvorijo popolni kvadrat.

Ocena časovne zahtevnosti algoritma temelji na predpostavki, da so števila C_i naključna. Zanima nas, koliko naključnih števil manjših od meje Y potrebujemo, da bomo dobili podmnožico števil, katere produkt bo kvadrat. Prvo nas zanima, kakšna je verjetnost, da je naključno naravno število, ki je manjše od števila X, Y-gladko. To nam pove funkcija $\psi(X, Y)$, ki je definirana kot:

$$\psi(X, Y) = \{\text{število } Y\text{-gladkih števil, ki so manjša kot } X\}$$

Verjetnost, da je naključno naravno število, ki je manjše od X, Y-gladko je $\frac{\psi(X,Y)}{X}$. Koliko naključnih števil, moramo pregledati, da najdemo Y-gladko število, je recipročno vrednosti: $\frac{X}{\psi(X,Y)}$. Ker potrebujemo približno $\pi(X)$ Y-gladkih števil, moramo pregledati približno $\frac{\pi(Y)\cdot X}{\psi(X,Y)}$ naključnih naravnih števil. Da ugotovimo, ali je število Y-gladko, potrebujemo $\pi(Y)$ korakov, zato je pričakovano število korakov

$$\frac{\pi(Y)^2 \cdot X}{\psi(X, Y)} \tag{3.6}$$

.

Poiskati želimo tak Y v odvisnosti od X, da bo minimiziral enačbo (3.6). Funkcija doseže minimum, ko je Y približno $e^{\frac{1}{2}\sqrt{\ln X \ln\ln X}}$. In kaj točno sta X in Y? Število X je približek za tipično pomožno število, ki ga algoritem izbere, in je približno $2\sqrt{n}$. Število Y pa je približek za največje praštevilo v bazi faktorjev. Zato je časovna zahtevnost algoritma $\mathcal{O}\left(e^{\sqrt{2\ln n \ln\ln n}}\right)$.

3.3.3 Shanksova faktorizacija s kvadratnimi formami (SQUFOF)

Shanksova faktorizacija s kvadratnimi formami (SQUFOF) je algoritem, ki ga je razvil Daniel Shanks in se odlično odreže pri faktorizaciji števil, ki so večja kot 10^{10} in manjša od 10^{18}. Algoritem uporablja binarne kvadratne forme za faktorizacijo števila n, vendar pa ga lahko predstavimo tudi z verižnimi ulomki, ki jih uporabimo za izračun vrednosti $\sqrt{n}$, ki lahko zapišemo kot neskončni verižni ulomek:

$$\sqrt{n} = q_0 + \cfrac{1}{q_1 + \cfrac{1}{q_2 + \ldots}} = [q_0, q_1, q_2, \ldots].$$

Verižni ulomek je periodičen, če obstaja $j > 0$, da velja $q_i = q_i + j$ za vse $i > 0$. Tedaj lahko zapišemo $\sqrt{n} = [q_0, \overline{q_1, \ldots q_j}]$, q_i imenujemo parcialni kvocient verižnega ulomka, racionalno število $[q_0, q_1, \ldots q_k]$ pa imenujemo k-ti približek verižnega ulomka. Definirajmo še

$$A_i = \begin{cases} 1, & \text{ko je } i = 0 \\ q_0, & \text{ko je } i = 1 \\ q_i \cdot A_{i-1} + A_{i-2}, & \text{ko je } i \geq 2 \end{cases}$$

in

$$B_i = \begin{cases} 0, & \text{ko je } i = 0 \\ 1, & \text{ko je } i = 1 \\ q_i \cdot B_{i-1} + B_{i-2}, & \text{ko je } i \geq 2. \end{cases}$$

Potem velja:

$$[q_0, q_1, \ldots q_k] = \frac{A_{k+1}}{B_{k+1}}, \quad \text{za } k \geq 0.$$

Definirajmo k-ti cel kvocient:

$$x_k = \begin{cases} \sqrt{n}, & \text{ko je } k = 0 \\ \dfrac{1}{x_{k-1} - q_{k-1}}, & \text{ko je } k \geq 1. \end{cases}$$

Da se pokazati, da je $x_k = \dfrac{P_k + \sqrt{n}}{Q_k}$ za $k \geq 0$, kjer sta P_k in Q_k definirana kot:

$$P_k = \begin{cases} 0, & \text{ko je } k = 0 \\ q_0, & \text{ko je } k = 1 \\ q_{k-1} \cdot Q_{k-1} - P_{k-1}, & \text{ko je } k \geq 2 \end{cases}$$

in

$$Q_k = \begin{cases} 1, & \text{ko je } k = 0 \\ n - q_0^2, & \text{ko je } k = 1 \\ Q_{k-2} + (P_{k-1} - P_k) \cdot q_{k-1}, & \text{ko je } k \geq 2. \end{cases}$$

Število q_k pa definiramo kot:

$$q_k = \begin{cases} \lfloor \sqrt{n} \rfloor, & \text{ko je } k = 0 \\ \left\lfloor \dfrac{q_0 + P_k}{Q_k} \right\rfloor, & \text{ko je } k \geq 1. \end{cases}$$

Kongruenca (3.5) ima tedaj rešitev

$$A_{k-1}^2 \equiv (-1)^k \cdot Q_k \mod n. \tag{3.7}$$

Do nje pridemo tako, da razširjamo $\sqrt{n}$, dokler ne najdemo kvadratnega števila $Q_k = R^2$ za sodi k. Potem sta števili $x \equiv A_{k-1}$ in $y \equiv R$ rešitvi kongruence (3.5). Če števili nista trivialni, lahko z Evklidovim algoritmom za n in $A_{k-1} \pm R$, poiščemo faktor števila n.

Algoritem števila A_i ne računa direktno, ampak hrani vrednosti $(-1)^k \cdot Q_k$, ki ne vodijo k trivialni rešitvi. Algoritem za izračun uporablja naslednje formule:

$$\begin{aligned} P_0 &= 0 \\ Q_0 &= 1 \\ Q_1 &= N - P_1^2 \\ q_i &= \left\lfloor \frac{\sqrt{n} + P_i}{Q_i} \right\rfloor && (3.8) \\ P_{i+1} &= q_i \cdot Q_i - P_i && (3.9) \\ Q_{i+1} &= Q_{i-1} + q_i \cdot (P_i - P_{i+1}) && (3.10) \end{aligned}$$

Algoritem 13 Shanksova faktorizacija s kvadratnimi formami

Vhod: število n
Izhod: faktor števila n oz. javi neuspeh (**failure**)

1: **if** n je kvadrat celega števila **then**
2: **return** $\sqrt{n}$
3: **end if**
4: **if** $n \equiv 1 \mod 4$ **then**
5: $D = 2n$
6: **else**
7: $D = n$
8: **end if**
9: $S = \lfloor \sqrt{d} \rfloor$
10: $Q_1 = 1$
11: $P = S$
12: $Q = D - P^2$
13: $L = \lfloor 2\sqrt{2\sqrt{D}} \rfloor$
14: $B = 2L$
15: $i = 1$
16: **while** $i \leq B$ **do** ▷ Cikel za iskanje ustrezne kvadratne forme
17: $q = \left\lfloor \frac{S+P}{Q} \right\rfloor$
18: $P_1 = q \cdot Q - P$
19: **if** $Q \leq L$ **and** Q je sodo število **then**
20: v QUEUE dodaj par $(\frac{Q}{2}, P \mod \frac{Q}{2})$
21: **else if** $Q \leq \frac{L}{2}$ **then**
22: v QUEUE dodaj par $(Q, P \mod Q)$
23: **end if**
24: $t = Q_1 + q(P - P_1)$
25: $Q_1 = Q$
26: $Q = t$
27: $P = P_1$
28: **if** i je sod **and** Q je kvadrat celega števila **then**
29: $r = \sqrt{Q}$
30: če v QUEUE ni para (r, t), za katerega velja $r \mid P - t$, pojdi na 40
31: **if** $r > 1$ **and** (obstaja par $(r, t) \in$QUEUE: $r \mid P - t$) **then**
32: iz QUEUE odstrani vse pare od prvega do vključno para (r, t)
33: $i = i + 1$
34: **end if**
35: **if** $r = 1$ **and** $(1, t) \in$QUEUE **then**
36: **return failure**
37: **end if**
38: **end if**
39: **end while** ▷ Našli smo kvadratno formo $F = (-Q_1, 2P, r^2)$

Če je vrednost $Q_i < 2\sqrt{2\sqrt{n}}$, ga shranimo v seznam, sicer ga zavržemo. Zanka algoritma se izvaja dokler ne najdemo $Q_{2i} = R^2$:

1. če je R (oz. $\frac{R}{2}$ če je R sod) v seznamu, nadaljujemo z razširjanjem, sicer
2. smo našli kvadrat, za katerega velja:

$$q_{2i} = \left\lfloor \frac{\sqrt{n} + P_{2i}}{Q_{2i}} \right\rfloor = \left\lfloor \frac{\sqrt{n} + P_{2i}}{R^2} \right\rfloor$$

Algoritem 14 Nadaljevanje: Shanksova faktorizacija s kvadratnimi formami (algoritem 13)

$Q_1 = r$ ▷ Poiskali bomo inverzni kvadratni koren forme F
$P = P + r \cdot \left\lfloor \frac{S - P}{r} \right\rfloor$
$Q = \frac{N - P^2}{Q_1}$
while true do ▷ Poiskali bomo faktor števila n
    $q = \left\lfloor \frac{S + P}{Q} \right\rfloor$
    $P_1 = q \cdot Q - P$
    if $P = P_1$ **then**
        break
    end if
    $t = Q_1 + q(P - P_1)$
    $Q_1 = Q$
    $Q = t$
    $P = P_1$
end while
if Q je sodo število **then**
    $Q = \frac{Q}{2}$
end if
return Q

Sedaj nadaljujemo z razširjanjem števila $\frac{\sqrt{n} - P_{2i}}{R}$. Najprej izračunamo

$$P'_0 = -P_{2i}, \quad Q'_0 = R, \quad Q'_i = \frac{n - P_i^2}{R},$$

nato pa uporabimo enačbe (3.8), (3.9) in (3.10). Postopek nadaljujemo, dokler ne najdemo P_j za katerega velja: $P_j = P_j + 1$. Če smo v prvem delu algoritma porabili k korakov, da smo našli $Q_{2i} = R^2$, bomo v drugem delu porabili $\frac{k}{2}$ korakov, da bomo našli $P_j = P_{j+1}$. Dobljeni Q_j (oz. $\frac{Q_j}{2}$ če je Q_j sod) je iskani faktor števila n. Časovna zahtevnost algoritma je $\mathcal{O}\left(\sqrt[4]{n}\right)$.

3.4 Številsko sito (NFS)

Leta 1988 je Pollard predstavil metodo, ki je omogočila hitro faktorizacijo Fermatovih števil in jo danes imenujemo posebno številsko sito (SNFS). Kasneje so jo posplošili, da omogoča faktorizacijo vseh števil ter jo poimenovali številsko sito (NFS). Z njim so leta 1996 uspeli faktorizirati 130-mestni izziv RSA-130, danes pa velja za najhitrejši algoritem za števila, pri katerih se drugi algoritmi najslabše obnesejo.

Številsko sito je izboljšava algoritma kvadratno sito (poglavje 3.2) - ko želimo faktorizirati veliko število n, moramo iskati gladka števila reda $\sqrt{n}$. Prednost številskega sita je v tem, da preiskuje manjša števila kot kvadratno sito in je zato večja verjetnost, da bodo števila gladka. Prav ta lastnost pripomore k učinkovitost algoritma. Pohitritev dosežemo z računanjem v polju oz. komutativnem obsegu, zato ga najprej definirajmo.

Definicija 3.6 (Obseg)**. Obseg** je množica $(\mathcal{O}, +, *)$ v kateri za poljubne elemente $a, b, c \in \mathcal{O}$ velja:

- komutativnost za operacijo $+$:

$$a + b = b + a$$

- asociativnost za operaciji $+$ in $*$:

$$(a + b) + c = a + (b + c)$$

$$(a * b) * c = a * (b * c)$$

- obstaja nevtralni element za operacijo $+$ (označimo ga z 0):

$$a + 0 = 0 + a = a$$

- obstaja nevtralni element za operacijo $*$ (označimo ga z 1 in ni enak nevtralnemu elementu za operacijo $+$):

$$a * 1 = 1 * a = a$$

- poljubni element $a \in \mathcal{O}$ ima nasprotni element $-a$, da velja:

$$a + (-a) = (-a) + a = 0$$

- za vsak $a \neq 0$ obstaja inverzni element a^{-1}, da velja:

$$a * a^{-1} = a^{-1} * a = 1$$

- distributivnost (z leve in desne strani), ki povezuje operaciji $+$ in $*$:

$$a * (b + c) = a * b + a * c$$

$$(a + b) * c = a * c + b * c$$

Definicija 3.7 (Komutativni obseg)**.** Obseg v katerem velja tudi komutativnost za operacijo $*$:

$$a * b = b * a,$$

imenujemo **komutativni obseg** oz. **polje**.

Množica

$$\mathbb{Q}[x] = \left\{ a_n x^n + a_{n-1} x^{n-1} + \ldots + a_1 x + a_0;\ a_n, a_{n-1}, \ldots, a_1, a_0 \in \mathbb{Q}, n \in \mathbb{N} \right\}$$

je množica vseh polinomov s koeficienti iz $\mathbb{Q}$. Izkaže se, da je $\mathbb{Q}[x]$ komutativen kolobar z enoto. Da zagotovimo, da je $\mathbb{Q}[x]$ res n-dimenzionalen prostor, mora biti $f(x)$ nerazcepen polinom.

Kakšna je razlika med kolobarjem in obsegom? Kolobar $(\mathcal{K}, +, *)$ nima definiranega inverznega elementa za operacijo $*$, vse ostale lastnosti pa so enake.

Metoda kvadratnega sita je hitra, ker deluje na kvadratnih ostankih po modulu števila, ki ga želimo faktorizirati, in ker s presejanjem ugotovi, kateri ostanki so gladka števila. Metoda bi bila še hitrejša, če bi bili dobljeni kvadratni ostanki manjši, saj bi se s tem povečala verjetnost, da so gladka števila. Izkaže se, da sploh ni potrebno, da so števila kvadratni ostanki, dovolj je, da so majhna.

Našo razlago metode bomo začeli s parom θ in $\phi(\theta)$, kjer θ leži v izbranem kolobarju, ϕ pa je homomorfizem iz kolobarja v $\mathbb{Z}_n$. Predpostavimo, da imamo k parov $(\theta_1, \phi(\theta_1)), \ldots (\theta_k, \phi(\theta_k))$. Produkt $\theta_1 \cdot \ldots \cdot \theta_k$ je kvadrat v izbranem kolobarju in ga označimo z γ^2. Z v^2 označimo celoštevilski kvadrat, ki zadošča kongruenci

$$\phi(\theta_1) \cdot \ldots \cdot \phi(\theta_k) \equiv v^2 \pmod{n}.$$

Če za neko celo število u velja $\phi(\gamma) \equiv u \pmod{n}$, velja tudi:

$$u^2 \equiv \phi(\gamma)^2 \equiv \phi(\gamma^2) \equiv \phi(\theta_1 \cdot \ldots \cdot \theta_k) \equiv \phi(\theta_1) \cdot \ldots \cdot \phi(\theta_k) \equiv v^2 \pmod{n}$$

Iz tega sledi

$$u^2 \equiv v^2 \pmod{n}.$$

To v praksi pomeni, da nam lahko $\gcd(u-v, n)$ pomaga faktorizirati število n.

Naj bo n sestavljeno število, ki ni potenca nekega števila in naj bo

$$f(x) = x^d + c_{d-1}x^{d-1} + \ldots + c_1 x + c_0$$

nerazcepen polinom v $\mathbb{Z}[x]$ s kompleksno ničlo α. Naj bo $\mathbb{Z}[\alpha]$ naš izbrani kolobar, ki si ga lahko predstavljamo kot množico urejenih d-terk: $\{(a_0, a_1, \ldots a_{d-1}), a_i \in \mathbb{Z}\}$, kjer vsaka d-terka predstavlja element $a_0 + a_1\alpha + \ldots + a_{d-1}\alpha^{d-1}$. Seštevanje in množenje dveh elementov kolobarja je enako kot seštevanje in množenje polinomov, le s to razliko, da moramo pri množenju paziti na stopnjo polinoma - če je prevelika, jo zmanjšamo z uporabo identitete $f(\alpha) = 0$.

Povezava med kolobarjem in našim številom n pride preko celega števila m za lastnostjo

$$f(m) \equiv 0 \pmod{n}.$$

Obstaja preprosta metoda za izbiro $f(x)$ in m. Za naš polinom najprej izberemo stopnjo d. Naj bo $m = \left\lfloor n^{\frac{1}{d}} \right\rfloor$ in zapišimo n v bazi m:

$$n = m^d + c_{d-1}m^{d-1} + \ldots + c_1 m + c_0,$$

kjer je vsak $c_i \in [0, m-1]$. Tako smo dobili polinom $f(x)$:

$$f(x) = x^d + c_{d-1}x^{d-1} + \ldots c_1 x + c_0.$$

Ni nujno, da je $f(x)$ nerazcepen polinom, saj če lahko $f(x)$ v $\mathbb{Z}[x]$ zapišemo kot produkt dveh polinomov $f(x) = g(x) \cdot h(x)$, potem lahko tudi n zapišemo kot produkt dveh netrivialnih števil: $n = g(m) \cdot h(m)$. Ker je faktorizacija polinomov relativno preprosta, se da $f(x)$ enostavno faktorizirati v $\mathbb{Z}[x]$. Če smo faktorizirali $f(x)$, smo uspešno faktorizirali tudi n, če pa je $f(x)$ nerazcepen, moramo nadaljevati z algoritmom.

Preden si pogledamo nadaljevanje algoritma, še definirajmo homomorfizem $\phi : \mathbb{Z}[x] \to \mathbb{Z}_n$, ki je definiran z $\phi(\alpha)$, ki je razred ostankov po modulu n. To pomeni, da ϕ najprej pretvori $a_0 + a_1\alpha + \ldots + a_{d-1}\alpha^{d-1}$ v celo število

$a_0+a_1m+\ldots+a_{d-1}m^{d-1}$, ki ga nato zmanjša po modulu n. Elementi kolobarja $\mathbb{Z}[\alpha]$, ki jih bomo obravnavali, bodo vsi oblike $a-b\alpha$, kjer sta $a, b \in \mathbb{Z}$ in velja $\gcd(a,b)=1$. Tako iščemo množico $\mathcal{S}$, ki vsebuje pare tujih celih števil (a,b), in zanjo velja:

$$\prod_{(a,b)\in\mathcal{S}} (a-b\alpha) = \gamma^2 \text{ za nek } \gamma \in \mathbb{Z}[\alpha],$$

$$\prod_{(a,b)\in\mathcal{S}} (a-bm) = v^2 \text{ za nek } v \in \mathbb{Z}.$$

Če je u tako celo število, da velja $\phi(\gamma) \equiv u \pmod{n}$, potem velja $u^2 \equiv v^2 \pmod{n}$ in lahko poskusimo poiskati faktor števila n z $\gcd(u-v,n)$.

Eksponentni vektorji

Zanima nas, kako poiskati množico $\mathcal{S}$, ki vsebuje pare (a,b). Pri metodi kvadratnega sita pregledamo interval z eno spremenljivko, pri številskem situ pa s presejanjem odkrivamo gladka števila danega polinoma ter jih povežemo z eksponentnimi vektorji. S pomočjo linearne algebre poiščemo elemente, za katere velja, da je njihov produkt kvadrat. Ker presejamo 2 števili (a in b), moramo za vsako vrednost števila a presejati vse vrednosti za b.

Definicija 3.8 (Eksponentni vektor)**.** Naj bo p_i oznaka i-tega praštevila in naj bo m dano število:

$$m = \prod_i p_i^{v_i}.$$

Potem je **eksponentni vektor** števila m vektor $v(m) = (v_1, v_2, \ldots)$.

Pojavi pa se vprašanje, katera števila presejati. Zaenkrat zanemarimo pogoj, da je $a-b\alpha$ kvadrat v $\mathbb{Z}[\alpha]$ in se osredotočimo na drugi pogoj za množico $\mathcal{S}$: produkt $a-bm$ naj bo kvadrat v $\mathbb{Z}$. Na začetku računanja smo izbrali m in mejo M, a in b pa pretečeta vsa pare celih števil $0 < |a|, b \leq M$. Tako dobimo $d-1$ homogenih polinomov $G(a,b) = a-bm$, ki jih presejamo, ko iščemo gladka števila. Pri tem odstranimo vse pare (a,b) za katere velja $\gcd(a,b) > 1$. Ko smo našli vsaj $\Pi(B)+1$ takih parov, lahko uporabimo linearno algebro na eksponentnih vektorjih, ki ustrezajo gladkim vrednostim $G(a,b)$, in poiščemo njihovo podmnožico $\mathcal{S}'$, za katero velja, da je produkt njenih elementov kvadrat.

Vendar pa smo pri tem zanemarili težji del problema: istočasno naj za pare $(a,b) \in \mathcal{S}$ velja, da je produkt $a-b\alpha$ kvadrat v $\mathbb{Z}[\alpha]$.

Naj bodo $\alpha_1 = a, \alpha_2, \ldots \alpha_d$ kompleksne ničle polinoma $f(x)$ in naj bo $\beta = s_0 + s_1\alpha + \ldots s_{d-1}\alpha^{d-1}$ element polja $\mathbb{Q}[\alpha]$ (s_i so racionalna števila). Njegova norma je definirana kot produkt ničel:

$$N(\beta) = \prod_{j=1}^{d}(s_0 + s_1\alpha_j + \ldots s_{d-1}\alpha_j^{d-1}).$$

Očitno je norma multiplikativna, kar pomeni

$$N(\beta\beta') = N(\beta) \cdot N(\beta').$$

Za nas je pomembna naslednja posledica: če je $\beta = \gamma^2$ za nek $\gamma \in \mathbb{Z}[\alpha]$, potem je $N(\beta)$ celoštevilski kvadrat, natančneje je kvadrat celega števila $N(\gamma)$.

Zato velja: če je produkt vrednosti $N(a - b\alpha)$, za $(a, b) \in \mathcal{S}$ kvadrat v $\mathbb{Z}$, je tudi produkt vrednosti $(a - b\alpha)$ kvadrat v $\mathbb{Z}[\alpha]$.

Prvo opazimo, da je

$$\begin{aligned} N(a - b\alpha) &= (a - b\alpha_1) \cdot (a - b\alpha_2) \cdot \ldots \cdot (a - b\alpha_d) \\ &= b^d \left(\tfrac{a}{b} - \alpha_1\right) \cdot \left(\tfrac{a}{b} - \alpha_2\right) \cdot \ldots \cdot \left(\tfrac{a}{b} - \alpha_d\right) \\ &= b^d \cdot f\left(\tfrac{a}{b}\right) \end{aligned}$$

saj je α ničla polinoma $f(x)$ in zato ga lahko zapišemo kot $f(x) = (x - \alpha_1) \cdot \ldots \cdot (x - \alpha_d)$. Naj bo $F(x, y)$ homogena oblika polinoma f:

$$F(x, y) = x^d + c_{d-1}x^{d-1}y + \ldots + c_0y^d = y^d \cdot f\left(\frac{x}{y}\right).$$

Potem velja $N(a - b\alpha) = F(a, b)$, kar pomeni, da lahko na $N(a - b\alpha)$ gledamo kot na funkcijo dveh spremenljivk: a in b.

Tako lahko zagotovimo, da je produkt $N(a - b\alpha)$ za $(a, b) \in \mathcal{S}$ kvadrat: števili a in b pretečeta $|a|, |b| \leq M$, s presejanjem poiščemo B-gladke vrednosti $F(a, b)$, generiramo ustrezne eksponentne vektorje in z matrično metodo poiščemo iskano podmnožico $\mathcal{S}'$ množice $\mathcal{S}$.

Ker pa želimo, da je tudi produkt vrednosti $a - bm$ hkrati kvadrat v $\mathbb{Z}$, spremenimo metodo tako, da išče gladke vrednosti produkta $F(a, b) \cdot G(a, b)$, ki je tudi polinom za a in b. Za gladke vrednosti kreiramo eksponentni vektor z dvema poljema koordinat: prvo polje ustreza praštevilski faktorizaciji za $F(a, b)$, drugo polje pa praštevilski faktorizaciji za $G(a, b)$. Tudi te večje vektorje zberemo v matriko in uporabimo linearno algebro za iskanje ustrezne podmnožice $\mathcal{S}'$. Pri tem opazimo, da sedaj za uspešno reševanje potrebujemo $2\Pi(B) + 3$ vektorjev v matriki (in ne le $\Pi(B) + 1$), saj ima vsak vektor

$2\Pi(B) + 2$ koordinat. V praksi to pomeni, da potrebujemo dvakrat več vektorjev, če želimo hkrati rešiti oba problema.

Več o tem, kako preverimo, ali je $(a - b\alpha)$ za $(a, b) \in \mathcal{S}$ kvadrat v $\mathbb{Z}[\alpha]$, si lahko bralec prebere v [CP00] - poglavje 6.2.4.

Ostane nam še vprašanje, kako naj poiščemo kvadratne korene za $\gamma \in \mathbb{Z}[\alpha]$ in $v \in \mathbb{Z}$. Obstaja več načinov, kako poiskati u - razlago nekaterih lahko bralec najde v poglavju 6.2.5 v [CP00].

Algoritem 15 Številsko sito (NFS)

Vhod: sestavljeno število n
Izhod: faktor števila n

1: $d = \left\lfloor \sqrt[3]{\frac{3 \ln n}{\ln \ln n}} \right\rfloor$
2: $B = \left\lfloor e^{\sqrt[3]{\frac{8}{9} \ln n (\ln \ln n)^2}} \right\rfloor$
3: $m = \left\lfloor n^{\frac{1}{d}} \right\rfloor$
4: $n = m^d + c_{d-1}m^{d-1} + \ldots c_1 m + c_0$ ▷ n zapišemo v bazi m
5: $f(x) = x^d + c_{d-1}x^{d-1} + \ldots + c_1 x + c_0$
6: Poskusi faktorizirati $f(x)$ v nerazcepne polinome v $\mathbb{Z}[x]$
7: **if** $f(x) = g(x) \cdot h(x)$ **then**
8: **return** $n = g(m) \cdot h(m)$
9: **end if**
10: $F(x, y) = x^d + c_{d-1}x^{d-1}y + \ldots c_0 y^d$
11: $G(x, y) = x - my$
12: **for** praštevilo $p_i < B$ **do**
13: sestavi množico $R(p_i) = \{r \in [0, p_i - 1] : f(r) \equiv 0 \pmod{p)_i}\}$
14: **end for**
15: $k = \lfloor 3 \log n \rfloor$
16: Izračunaj prvih k takih praštevil $q_1, q_2, \ldots q_k > B$, da $R(q_j)$ vsebuje element s_j za katerega velja $f'(s_j) \not\equiv 0$ in shrani k parov (q_j, s_j)
17: $B' = \sum_{p \leq B} \#R(p)$
18: $V = 1 + \Pi(B) + B' + K$
19: $M = B$
20: S sitom poišči množico $\mathcal{S}$ tujih celoštevilskih parov (a, b): $0 < |a|, b \leq M$ in $F(a, b) \cdot G(a, b)$ naj bo B-gladko število. Išči dokler $\#\mathcal{S} > V$, če ne uspeš, povečaj M in poskusi znova oz. pojdi na 1. korak in povečaj B.

Algoritem 16 Nadaljevanje: številsko sito (algoritem 15)

for $(a, b) \in \mathcal{S}$ **do** ▷ Zgradili bomo matriko velikosti $V \times \#\mathcal{S}$
  $\vec{v}$ je ničelni vektor dolžine V
  if $G(a, b) < 0$ **then**
    $\vec{v}[1] = 1$ ▷ Določimo 1. komponento vektorja $\vec{v}$
  end if
  $|G(a, b)| = \prod_{i=1}^{\pi(B)} p_i^{\gamma_i}$
  for $2 \leq i \leq \Pi(B) + 1$ **do**
    if γ_i je lih **then**
      $\vec{v}[i] = 1$
    end if
  end for
  $i = 1$
  for vse pare $(p, r) : r \in R(p)$ **do**
    if $\vec{v}_{p,r}(a - b\alpha)$ je lih **then**
      $\vec{v}[\Pi(B) + 1 + i] = 1$
    end if
    $i = i + 1$
  end for
  for $1 \leq i \leq k$ **do**
    if $\left(\frac{a - bs_i}{q_i}\right) = -1$ **then**
      $\vec{v}[\Pi(B) + 1 + B' + i] = 1.$
    end if
  end for
  Vstavi eksponentni vektor $\vec{v}(a - b\alpha)$ kot naslednjo vrstico matrike.
end for
Z neko metodo linearne algebre poišči neprazno podmnožico $\mathcal{S}'$ množice $\mathcal{S}$, da je $\sum_{(a,b)\in\mathcal{S}'} \vec{v}(a - b\alpha)$ ničelni vektor po modulu 2.
Uporabi znano faktorizacijo celoštevilskega kvadrata $\prod_{(a,b)\in\mathcal{S}'} (a - bm)$ za iskanje ostanka $v \mod n$ z

$$\prod_{(a,b)\in\mathcal{S}'} (a - bm) \equiv v^2 \pmod{n}$$

Z neko metodo poišči kvadratni koren γ v $\mathbb{Z}[\alpha]$ za $f'(a)^2 \prod_{(a,b)\in\mathcal{S}'} (a - b\alpha)$ in z enostavno zamenjavo $\alpha \to m$ izračunaj $u = \phi(\gamma) \pmod{n}$
return $\gcd(u - f'(m)v, n)$

Časovna zahtevnost

Gotovo se nam porodi vprašanje, zakaj je ta algoritem hiter pri faktorizaciji števil. Izkaže se, da je ključna velikost pomožnih števil, ki jih želimo sestaviti v kvadrat in jih bomo označili z X. Ko poznamo njihovo velikost, poznamo tudi časovno zahtevnost algoritma. Pri metodi kvadratnega sita je $X \sim n^{\frac{1}{2}+\varepsilon}$, pri algoritmu NFS pa lahko izberemo polinomski $f(x)$ in celo število m tako, da je produkt $F(a,b) \cdot G(a,b)$ omejen z vrednostjo X, ki je oblike $e^{c\sqrt[3]{\ln^2 n \ln\ln n}}$.

Tako je moč množice, ki jo presejamo, približno $\frac{2}{3}$ moči števk števila n. Pri kvadratnem situ je moč množice večja kot $\frac{1}{2}$ števk števila n. Tako je asimptotična časovna zahtevnost algoritma NFS

$$\mathcal{O}\left(e^{c\sqrt[3]{\ln n(\ln\ln n)^2}}\right).$$

Vrednost konstante c je odvisna od vrste algoritma: za splošno številsko sito (NFS) je $c = \sqrt[3]{\frac{64}{9}} = 1,923$, za posebno številsko sito (SNFS) pa je $c = \sqrt[3]{\frac{32}{9}} = 1,523$.

3.4.1 Primerjava algoritmov NFS in QS

Kvadratno sito je hitrejši algoritem za manjša, do 100-mestna števila, za večja števila (večja kot 130-mestna števila) pa je boljše številsko sito. Seveda je veliko odvisno tudi od implementacije algoritmov - številsko sito je bolj kot kvadratno sito občutljiv na količino spomina, ki ga ima računalnik.

3.4.2 Posebno številsko sito (SNFS)

Posebno številsko sito je izpeljava številskega sita za tista števila, kjer lahko uporabimo posebno ugodne polinome - to so polinomi z majhnimi koeficienti. SNFS je tako primeren za števila oblike $r^e \pm s$, kjer sta r in s majhna.

Postopek faktorizacije je zelo podoben splošnemu številskemu situ. Razlikuje se predvsem v parametrih, saj ni vsako število primerno za SNFS. Vnaprej moramo poznati polinom f z majhnimi koeficienti in stopnjo d - izkaže se, da je optimalna stopnja polinoma

$$d = \sqrt[3]{3 \cdot \frac{\ln n}{\ln\ln n}}.$$

Poznati moramo tudi vrednost x, za katero velja: $f(x) \equiv 0 \pmod{n}$ in mora zadoščati kongruenci $ax + b \equiv 0 \pmod{n}$ za $a, b \leq \sqrt[d]{n}$.

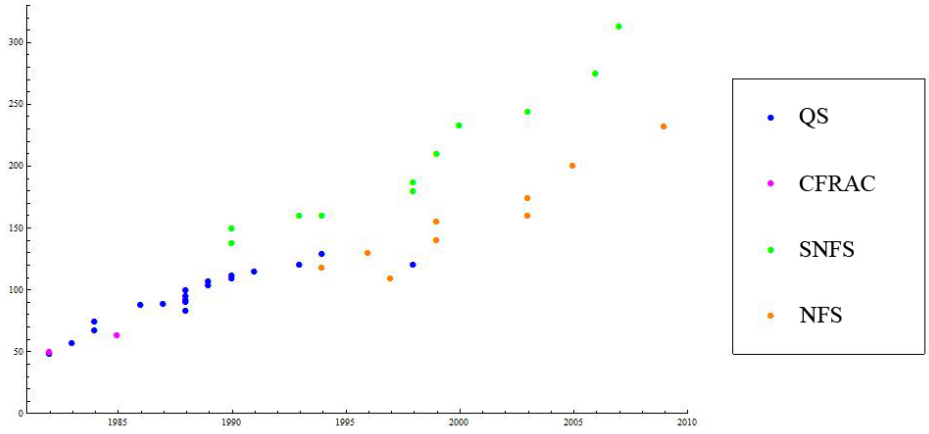
Takšni polinomi obstajajo za števila oblike $a^b \pm 1$, Fibonaccijeva števila in Lucasova števila. Največji uspeh algoritma SNFS je faktorizacija 180-mestnega Cunninghamovega števila.

Poglavje 4

Zaključek

V diplomski nalogi smo si ogledali 12 algoritmov za faktorizacijo naravnih števil in potrebno matematično podlago za njihovo razumevanje. V tabeli 4.1 so zbrani vsi predstavljeni algoritmi in njihove časovne zahtevnosti. Pri tem naj bralca opozorimo, da je večina podanih časovnih zahtevnosti le hevristično ocenjenih, saj natančne časovne zahtevnosti ne znamo določiti.

Najhitrejši algoritem je številsko sito, ki za faktorizacijo števila n porabi $\mathcal{O}\left(e^{c\sqrt{\ln n(\ln\ln n)^2}}\right)$ časa. Z njim so decembra 2009 faktorizirali trenutno največje faktorizirano število - 232-mestno število izziva RSA-768. Izmed števil, ki jih lahko faktoriziramo s posebnim številskim sitom, pa je največje 313-mestno število $2^{1039}-1$, ki so ga faktorizirali leta 2007. Napredek pri faktorizaciji naravnih števil je predstavljen na sliki 4.1.



Slika 4.1: Največja faktorizirana števila skozi leta.

Algoritem	**Časovna zahtevnost**
Poskusno deljenje	$\mathcal{O}\left(\sqrt{n}\right)$
Pollardov $p-1$ algoritem	$\mathcal{O}\left(B \log B (\log n)^2 + (\log n)^3\right)$
Williamsov $p+1$ algoritem	$\mathcal{O}\left(q\right)$
Lenstrova metoda (ECM)	$\mathcal{O}\left(e^{(\sqrt{2}+\mathcal{O}(1))\sqrt{\ln p \ln \ln p}}\right)$
Pollardov ρ algoritem	$\mathcal{O}\left(\sqrt[4]{n}\right)$
Fermatov algoritem	$\mathcal{O}\left(\sqrt{n}\right)$
Eulerjev algoritem	$\mathcal{O}\left(n^{\frac{1}{3}+\varepsilon}\right)$
Dixonov algoritem	$\mathcal{O}\left(e^{(\sqrt{2}+\mathcal{O}(1))\sqrt{\ln n \ln \ln n}}\right)$
Kvadratno sito (QS)	$\mathcal{O}\left(e^{\sqrt{\ln n \ln \ln n}}\right)$
Algoritem za faktorizacijo z verižnimi ulomki (CFRAC)	$\mathcal{O}\left(e^{\sqrt{2 \ln n \ln \ln n}}\right)$
Shanksova faktorizacija s kvadratnimi formami (SQUFOF)	$\mathcal{O}\left(\sqrt[4]{n}\right)$
Številsko sito (NFS)	$\mathcal{O}\left(e^{c \sqrt[3]{\ln n (\ln \ln n)^2}}\right)$

Tabela 4.1: Pregled časovnih zahtevnosti predstavljenih algoritmov

Zaradi omejenega razvoja kvantnih računalnikov, se je pozornost v teh letih usmerila v paralelne algoritme. Večino predstavljenih algoritmov so že prilagodili paralelizmu, zainteresiran bralec pa si lahko več o tem prebere v članku [Br99].

Možnosti za nadaljnje delo vidim predvsem v vsebinah, ki jih zaradi preobsežnosti področja v tej diplomski nalogi nismo predstavili - kvantnih algoritmih (Shorov algoritem) ter algoritmih, ki jih lahko izvajamo na paralelnih računalnikih.

Dodatek A

Implementacija algoritmov

Datoteka Faktorizacija.java

```
import java.util.*;
import java.io.*;

public class Faktorizacija {
   public static void main(String[] args){
      double B, c, n, B1;
      Scanner in = new Scanner(System.in);
      System.out.println("FAKTORIZACIJA Z ALGORITMI POSKUSNO DELJENJE, ↩
          POLLARDOV (p-1) ALGORITEM IN SQUFOF.");
      System.out.println("Enkratna izvedba (1) ali zanka (2)?");
      try{
         int izbira = Integer.parseInt(in.nextLine());
         if (izbira==1){
            /*
             * Enkratna izvedba zanke
             * Potrebujemo naslednje parametre: število n, mejo B, osnovo c.
             */
            System.out.println("Vnesi argumente!");
            System.out.println("[število n], [zgornja meja B], [osnova c]");
            System.out.println("Vpiši število n:");
            n=Double.parseDouble(in.nextLine());
            System.out.println("Vpiši zgornjo mejo B:");
            B=Double.parseDouble(in.nextLine());
            System.out.println("Vpiši osnovo c:");
            c=Double.parseDouble(in.nextLine());
            in.close();
            enkratna_izvedba(n, B, c);
         }
         else if(izbira == 2){
            /*
             * Zanka med spodnjo mejo B1 in zgornjo mejo B za vse algoritme
             * Potrebujemo naslednje parametre: spodnja meja B1, zgornja ↩
                 meja B, osnova c.
             */
            System.out.println("Vnesi argumente!");
            System.out.println("[spodnja meja za zanko B1], [zgornja meja B↩
```

```
                ], [osnova c]");
            System.out.println("Vpiši spodnjo mejo za zanko:");
            B1=Double.parseDouble(in.nextLine());
            System.out.println("Vpiši zgornjo mejo B:");
            B=Double.parseDouble(in.nextLine());
            System.out.println("Vpiši osnovo c:");
            c=Double.parseDouble(in.nextLine());
            in.close();
            zanka(B1, B, c);
        }
        else{
            System.err.println("Napačen parameter!");
        }
    }
    catch (Exception e){
        System.err.println("Napačen vnos parametra!");
    }
}

public static void enkratna_izvedba(double n, double B, double c){
    /*
     * Algoritem: poskusno deljenje
     */
    LinkedList<Double> trialDivision = TrialDivision.algorithm(n);
    StringBuffer zapis = new StringBuffer(n+" "+TrialDivision.↩
        potencniZapis(trialDivision)+" "+TrialDivision.time);
    System.out.println("Poskusno deljenje: "+zapis);

    /*
     * Algoritem: Pollardov (p-1) algoritem
     */
    double faktor=Pollard.algorithm((double)n, B, c);
    zapis = new StringBuffer();
    if (faktor == -1){
        /*
         * Metoda Pollard ni uspela faktorizirati podanega števila n
         */
        zapis = new StringBuffer(n+" "+n+" "+Pollard.time);
    }
    else{
        zapis = new StringBuffer(n+" "+faktor+"*"+n/faktor+" "+Pollard.time↩
            );
    }
    System.out.println("Pollardov (p-1) algoritem: "+zapis);

    /*
     * Algoritem: SQUFOF
     */
    faktor = Squfof.algoritem(n);
    zapis = new StringBuffer();
    if(faktor == -1){
        /*
         * Algoritem ni uspel faktorizirati podano število n
         */
        zapis = new StringBuffer(n+" Algoritem ni našel faktorja. "+Pollard↩
            .time);
    }
    else{
        zapis = new StringBuffer(n+" "+faktor+"*"+n/faktor+" "+Pollard.time↩
```

```
            );
        }
        System.out.println("SQUFOF: "+zapis);
    }

    public static void zanka(double B1, double B, double c){

        /*
         * Algoritem: poskusno deljenje
         */
        try{
            FileWriter stream = new FileWriter("TrialDivision.txt");
              BufferedWriter zapisovanje = new BufferedWriter(stream);

              /*
               * Zanka for, ki generira števila, ki jih poskusimo faktorizirati↩
                   z algoritmom poskusno deljenje
               */
            for(double n=B1; n<=B; n++){
                /*
                 * Klic metode TrialDivision (algoritem Poskusno deljenje)
                 */
                LinkedList<Double> trialDivision = TrialDivision.algorithm(n);
                /*
                 * priprava izpisa v datoteko
                 */
                StringBuffer zapis = new StringBuffer(n+" "+TrialDivision.↩
                    potencniZapis(trialDivision)+" "+TrialDivision.time);

                /*
                 * Zapis v datoteko TrialDivision.txt
                 * oblika zapisa vrstice: "faktorizirano število n" "potenčni ↩
                     zapis faktorizacije" "čas potreben za faktorizacijo"
                 */
                zapisovanje.write(zapis.toString());
                zapisovanje.newLine();
            }
             zapisovanje.close();
        }
        /*
         * Izpis morebitne napake
         */
        catch (Exception e){
            System.err.println("Napaka: " + e.getMessage());
        }

        /*
         * Algoritem: Pollardov (p-1) algoritem
         */
        try{
            FileWriter stream = new FileWriter("Pollard(p-1).txt");
              BufferedWriter zapisovanje = new BufferedWriter(stream);

              /*
               * Zanka for, ki generira števila, ki jih poskusimo faktorizirati↩
                   s Pollardovim (p-1) algoritmom
               */
              for(double n=B1; n<=B; n++){
                /*
```

```
         * klic metode Pollard (Pollardov (p-1) algoritem)
         */
        double g=Pollard.algorithm((double)n, B, c);
        StringBuffer zapis = new StringBuffer();
        if (g==-1){
           /*
            * Metoda Pollard ni uspela faktorizirati podanega števila n
            */
           zapis = new StringBuffer(n+" "+n+" "+Pollard.time);
        }
        else{
           zapis = new StringBuffer(n+" "+g+"*"+n/g+" "+Pollard.time);
        }

        /*
         * Zapis v datoteko TrialDivision.txt
         */
        zapisovanje.write(zapis.toString());
        zapisovanje.newLine();
     }
      zapisovanje.close();
}
/*
 * Izpis morebitne napake
 */
catch (Exception e){
   System.err.println("Napaka: " + e.getMessage());
}


/*
 * Algoritem: Shanksova faktorizacija s kvadratnimi formami (SQUFOF)
*/
try{
   FileWriter stream = new FileWriter("SQUFOF.txt");
     BufferedWriter zapisovanje = new BufferedWriter(stream);

     /*
      * Zanka for, ki generira števila, ki jih poskusimo faktorizirati↩
          s SQUFOF
      */
     for(double n=B1; n<=B; n++){
      /*
       * klic metode Squfof (Pollardov (p-1) algoritem)
       */
      double faktor = Squfof.algoritem(n);
      StringBuffer zapis = new StringBuffer();
      if(faktor == -1){
         /*
          * Algoritem ni uspel faktorizirati podano število n
          */
         zapis = new StringBuffer(n+" Algoritem ni našel faktorja. "+↩
             Pollard.time);
      }
      else{
         zapis = new StringBuffer(n+" "+faktor+"*"+n/faktor+" "+↩
             Pollard.time);
      }
```

```
            /*
             * Zapis v datoteko SQUFOF.txt
             */
            zapisovanje.write(zapis.toString());
            zapisovanje.newLine();
         }
          zapisovanje.close();
      }
      /*
       * Izpis morebitne napake
       */
      catch (Exception e){
         System.err.println("Napaka: " + e.getMessage());
      }

   }

   /*
    * Preprosta metoda za izračun največjega skupnega delitelja števil a in ↩
      b
    */
   public static double GCD(double a, double b)
   {
      if (b==0) return a;
      return GCD(b,a%b);
   }
}
```

Datoteka TrialDivision.java

```
import java.util.*;

public class TrialDivision {
   static double time;

   /*
    * Algoritem: poskusno deljenje
    * Metoda vrne LinkedList<Double> F, ki vsebuje vse faktorje števila n. ↩
      Kot vhod ima podano število n, ki ga želimo faktorizirati.
    */
   public static LinkedList<Double> algorithm(double n){
      time = System.currentTimeMillis();
      LinkedList<Double> F = new LinkedList<Double>();
      while(n%2==0){
         n=n/2;
         F.add(2.0);
      }
      double d=3;
      while(d*d<=n){
         while(n%d==0){
            n=n/d;
            F.add(d);
         }
         d+=2;
```

```
        }
        if(n==1){
            time = System.currentTimeMillis() - time;
            return F;
        }
        else{
            F.add(n);
            time = System.currentTimeMillis() - time;
            return F;
        }
    }

    /*
     * Metoda, ki oblikuje potenčni izpis strukture LinkedList<Double>: p_1^↩
       a_1*p_2^a_2*...
     */
    public static String potencniZapis(LinkedList<Double> trialDivision){
        StringBuffer s = new StringBuffer();
        int indeks=1;
        int dolzina=trialDivision.size();
        double trenutni=0;
        double naslednji=0;
        if(dolzina>1){
            for (int i=0; i<dolzina-1; i++){
                trenutni = trialDivision.get(i);
                naslednji = trialDivision.get(i+1);
                if (trenutni==naslednji){
                    indeks++;
                }
                else{
                    s.append((int)trenutni);
                    if (indeks!=1){
                        s.append("^"+indeks);
                    }
                    indeks=1;
                    s.append("*");
                }
            }
            if (trenutni==naslednji){
                s.append((int)trenutni+"^"+indeks);
            }
            else{
                s.append((int)naslednji);
            }
        }
        else{
            trenutni = trialDivision.get(0);
            s.append((int)trenutni+"*1");
        }
        return s.toString();
    }
}
```

Datoteka `Pollard.java`

```
public class Pollard {
   static long time = 0;

   /*
    * Algoritem: Pollard (p-1) algoritem
    * Metoda vrne število g, ki je faktor števila n. Kot vhod metoda ↩
       potrebuje število n, ki ga želimo faktorizirati, mejo B in osnovo c.
    */

   public static double algorithm(double n, double b, double c){
      time = System.currentTimeMillis();
      double g = -1;
      double m = c;
      for(int i=1; i<=b; i++){
         int st=1;
         double tmp=m;
         while(st<=i){
            tmp*=m;
            tmp=tmp%n;
            st++;
         }
         m=tmp;
         g=Faktorizacija.GCD(m-1,n);
         if (g<n && g>1){
            break;
         }
      }
      if (g==1||g==n){
         g=-1;
      }
      time = System.currentTimeMillis() - time;
      return g;
   }
}
```

Datoteka Squfof.java

```
import java.util.*;

public class Squfof {

   /*
    * Algoritem: Shanksova faktorizacija s kvadratnimi formami (SQUFOF)
    * Metoda vrne število Q, ki je faktor števila n. Kot vhod metodi podamo ↩
       število n, ki ga želimo faktorizirati.
    */
   public static double algoritem(double n){
      /*
       * Inicializacija
       */
      LinkedList<Par> Queue = new LinkedList<Par>();
      int koren = (int)Math.sqrt(n);
      if((koren*koren)==n){
         return Math.sqrt(n);
```

```
}
double D;
if(n%4==1){
   D=2*n;
}
else{
   D=n;
}
double S=(double)((int)Math.sqrt(D));
double Q1=1;
double P=S;
double Q=D-P*P;
double L=(int)(2*Math.sqrt(2*Math.sqrt(D)));
double B=2*L;
int i=0;
double r=0;
double q=0;
double P1=0;
double t=0;

boolean nasli=true;

/*
 * Cikel za iskanje ustrezne kvadratne forme
 */
while(i<=B && nasli){
   q=(int)((S+P)/Q);
   P1=q*Q-P;
   if (Q<=L && Q%2==0){
      Par par = new Par(Q/2, P%(Q/2));
      Queue.add(par);
   }
   else if (Q<=L/2){
      Par par = new Par(Q, P%Q);
      Queue.add(par);
   }
   t=Q1+q*(P-P1);
   Q1=Q;
   Q=t;
   P=P1;
   koren=(int)Math.sqrt(Q);
   if(i%2==0 && (koren*koren==Q)){
      r=Math.sqrt(Q);

      /*
       * Pregledamo seznam Queue - iščemo par (r,t): r|P-t
       */
      Iterator<Par> iterator = Queue.iterator();
      nasli = false;
      while(iterator.hasNext() && nasli==false){
         Par par = (Par) iterator.next();
         if (par.get_x()==r && par.get_y()==t && ((P-t)%r==0)){
            nasli = true;
            if (r>1){
               /*
                * Iz Queue odstrani vse pare do vključno (r,t)
                */
               Queue.indexOf(par);
               while(!(Queue.getFirst().equals(par))){
```

```
                    Queue.removeFirst();
                }
                Queue.removeFirst();
                i++;
            }
            if(r==1){

            }
        }
    }
    if(r==1){
        Par par = new Par(1,t);
        if (Queue.contains(par)){
            return (0);
        }
    }
}
else{
    i++;
}
}
Q1=r;
P=P+r*((int)((S-P)/r));
Q=(int)((n-P*P)/Q1);

boolean zanka=true;
int j=1;
double[] tabela_P1 = new double[(int)B];
double[] tabela_P = new double[(int)B];
double[] tabela_Q1 = new double[(int)B];
double[] tabela_Q = new double[(int)B];
double[] tabela_q = new double[(int)B];
tabela_P1[0]=P1;
tabela_P[0]=P;
tabela_Q1[0]=Q1;
tabela_Q[0]=Q;
tabela_q[0]=0;
/*
 * Iščemo faktor števila n
 */
while(zanka && j<B){
    tabela_Q1[j]=tabela_Q[j-1];
    tabela_q[j]=(int)((S+tabela_P[j-1])/tabela_Q1[j]);
    tabela_P1[j]=tabela_q[j]*tabela_Q[j-1]-tabela_P[j-1];
    tabela_Q[j]=tabela_Q1[j-1]+tabela_q[j]*(tabela_P[j-1]-tabela_P1[j])↩
        ;
    tabela_P[j]=tabela_P1[j];

    if(tabela_P[j-1]==tabela_P1[j]){
        zanka=false;
        break;
    }
    j++;
}

if(zanka==false){
    Q=tabela_Q1[j];
    if(tabela_Q1[j]%2==0){
        Q=tabela_Q1[j]/2;
```

```
            }
            return Q;
        }
        else{
            return -1;
        }
    }

}
```

Datoteka `Par.java`

```
public class Par {
    double x;
    double y;
    public Par(double x, double y){
        x=this.x;
        y=this.y;
    }
    public double get_x(){
        return x;
    }
    public double get_y(){
        return y;
    }
}
```

Stvarno kazalo

algoritem za faktorizacijo z verižnimi ulomki, 26, 28, 46, 62

baza faktorjev, 36, 47

Dixonov algoritem, 33, 34, 62

eksponentni vektor, 55, 58
eliptična krivulja, 19
Eulerjev algoritem, 28, 62
Eulerjev kriterij, 15

faktorizacija, 5
Fermat
  Fermatov algoritem, 26, 30, 33, 62
  mali Fermatov izrek, 14

gladko število, 6, 35, 47, 52
grupa, 20
  grupa na eliptični krivulji po modulu p, 22

kvadratni ostanek po modulu p, 15, 17, 36

Legendrov simbol, 16, 35
Lenstrova metoda, 22, 62
Lucasova zaporedja, 16

največji skupni delitelj, 6

obseg, 52
  Galaisov obseg, 39
  komutativni obseg, 53

polje, *glej* komutativni obseg
Pollard, 52
  Pollardov ρ algoritem, 24, 62
  Pollardov $p-1$ algoritem, 13, 15, 62
poskusno deljenje, 10, 19, 27, 28, 33, 38, 47, 62

razširjen Evklidov algoritem, 6, 15, 44
RSA, 8
  izziv RSA, 9, 35, 52, 61

Shanksova faktorizacija s kvadratnimi formami, 28, 48, 62
Shorov algoritem, 9, 62
sito
  Eratostenovo sito, 7, 34
  kvadratno sito, 24, 26, 35, 46, 52, 59, 62
  kvadratno sito na družini polinomov, 42
  posebno številsko sito, 52, 59
  številsko sito, 24, 52, 61, 62

verižni ulomek, 43, 48

Williamsov $p+1$ algoritem, 15, 62

Seznam algoritmov

1 Razširjen Evklidov algoritem 6
2 Eratostenovo sito 7
3 Poskusno deljenje 12
4 Pollardov $p-1$ algoritem 14
5 Williamsov $(p+1)$ algoritem 18
6 Lenstrova metoda (ECM) 23
7 Pollardov ρ algoritem 26
8 Fermatov algoritem 27
9 Eulerjev algoritem 31
10 Dixonov algoritem 35
11 Kvadratno sito 42
12 Bháscara-Brounckerjev algoritem 45
13 Shanksova faktorizacija s kvadratnimi formami 50
14 Nadaljevanje: Shanksova faktorizacija s kvadratnimi formami (algoritem 13) 51
15 Številsko sito (NFS) 57
16 Nadaljevanje: številsko sito (algoritem 15) 58

Literatura

[Br89] D. M. Bressoud, Factorization and Primality Testing, Springer-Verlag, 1989.

[CP00] R. Crandall, C. Pomerance, *Prime Numbers: A Computational Perspective*, Springer, 2000.

[Co93] H. Cohen, *A Course in Computational Algebraic Number Theory*, Springer-Verlag, 1993.

[Ri94] H. Riesel, *Prime Numbers and Computer Methods for Factorization*, Birkhäuser, 1994.

[St06] D. R. Stinson, *Cryptography: Theory and Practice*, Chapman & Hall/CRC, 2006

[Or88] O. Ore, *Number theory and its history*, Courier Dover Publications, 1988.

[Po96] C. Pomerance, "A Tale of Two Sieves," v *American Mathematical Society* št. 43, str. 1473-1485, 1996

[Mc96] J. McKee, "Turning Euler's Factoring Method into a Factoring Algorithm," v *Bulletin of the London Mathematical Society*, št. 28, str. 351-355, 1996.

[Br99] Richard P. Brent, "Parallel Algorithms for Integer Factorisation," dostopen na http://wwwmaths.anu.edu.au/ brent/pd/rpb115.pdf

[Sh94] P. W. Shor, "Algorithms for Quantum Computation: Discrete Logarithms and Factoring," v *35th Annual Symposium on Foundations of Computer Science: proceedings*, str. 124-134, 1994.

[Mo02] F. Morain, "Thirty Years of Integer Factorization," v *Proceedings of the Algorithms Seminar 2000 - 2001*, str. 77-80, 2002.