Algoritmi za faktorizacijo celih števil

Matej Marinko

Faculty of Computer and Information Science
Večna pot 113
1000 Ljubljana

Abstract. V članku razložimo pomen problema faktorizacije celih števil in si ogledamo nekaj algoritme za faktorizacijo. Pri posameznih algortimih opišemo idejo v ozadju ter zakaj le-ti algoritmi sploh delujejo. Predstavljene algoritme tudi implementiramo.

Key words: integer factorization

1. UVOD

Problem faktorizacije celih števil je že zelo star, vemo da so praštevila poznali že stari Grki. Kljub temu je ostal pomemben tudi v sodobnem svetu, saj varnost šifrirnega algoritma RSA med drugim temelji na tem, da je pri določenih pogojih ta problem težek. Poznamo mnogo algoritmov za faktorizacijo, ki temeljijo na različnih lastnostih naravnih števil. V nadaljevanju si bomo ogledali poskušanje, Fermatov algoritem, faktorizacijo z vzpenjanjem, Pollardovo $p-1$ metodo, Lenstrin algoritem ter kvadratno sito.

Najprej bomo natančneje definirali problem, nato pa naredili predgled različnih algoritom za faktorizacijo, kjer bomo poleg samega algoritma predstavili tudi njegovo časovno zahtevnost in uporabnost v praksi. Začeli bomo s preprosto metodo poskušanja, nadaljevali pa z vedno bolj zahtevnimi, pri čemer ne velja nujno, da je zahtevnejša metoda boljša. Vse predstavljene algortime tudi implementiramo v programskem jeziku Python.

2. OPIS PROBLEMA

Spomnimo se zakaj velja, da je RSA šifriranje varno. Naj bosta p in q veliki praštevili in $n = pq$. Naj bo e šifrirni eksponent in d dešifrirni eksponent, tako da velja $ed \equiv 1 \pmod{\varphi(n)}$. Par (n, e) predstavlja javni ključ. Varnost RSA temelji na domnevi, da so naslednji problemi težki:

- Odšifriranje brez zasebnega ključa je težko: iz $x^e \bmod n$ je težko izračunati x v doglednem času.
- Iz javnega ključa (n, e) je težko izračunati zasebni eksponent d v doglednem času.
- Ne moremo faktorizirati števila $n = pq$ v doglednem času.

Osredotočili se bomo na zadnji problem. V primeru, da nam iz produkta n uspe izračunati faktorja p in q, lahko izračunamo $\varphi(n) = (p-1)(q-1)$, nato pa z razširjenim Evklidovim algoritmom izračunamo še šifrirni eksponent $d \equiv e^{-1} \pmod{\varphi(n)}$. V nadaljevanju si bomo ogledali različne algoritme za faktorizacijo naravnih števil.

Definicija 2.1. Naj bo n sestavljeno število. Če najdemo število $f \in \mathbb{N}$, ki deli n, rečemo, da smo rešili *faktorizacijski problem* števila n, saj lahko n zapišemo kot $n = f \cdot m$ za nek $m \in \mathbb{N}$ in s tem faktoriziramo n.

V primeru faktorizacije modula n iz RSA zadostuje, da najdemo en sam faktor, saj je n sestavljen iz dveh praštevil. Če pa imamo poljubno sestavljeno število

$$n = p_1^{e_1} p_2^{e_2} \cdots p_k^{e_k},$$

za neka različna praštevila $p_1, \ldots, p_k$ in eksponente $e_1, \ldots e_k \in \mathbb{N}$ ter nam uspe n zapisati kot produkt $n = f \cdot m$, potem postopek rekurzivno ponovimo na vsakem izmed faktorjev. Na koncu dobimo celotno faktorizacijo števila n.

Algoritme za faktorizacijo lahko v grobem razdelimo v dve skupini:

- *Splošni algoritmi za faktorizacijo*, katerih čas izvajanja je odvisen predvsem od velikosti števila n, ki ga želimo faktorizirati in ne od faktorjev, ki sestavljajo n. Primer algoritma, ki spada v to skupino je faktorizacija s številskim rešetom (angl. number field sieve).
- *Posebni algoritmi za faktorizacijo*, katerih čas izvajanja je odvisen predvsem od velikosti faktorja f, ki ga uspemo najti. Primeri algoritmov so "poskušanje", Pollardova ρ metoda in Lenstrin algoritem.

2.1. Eliptične krivulje

Preden se dokončno posvetimo algoritmom, se spomnimo grupe na eliptični krivulji nad končnim obsegom

$\mathbb{Z}_p$, saj bomo to teorijo potrebovali v nadaljevanju pri Lenstrinem algoritmu.

Definicija 2.2. Pri nekem praštevilu p je za dani števili a in b *eliptična krivulja* E nad $\mathbb{Z}_p$ množica točk:

$$E = \left\{(x, y) \in \mathbb{Z}_p \times \mathbb{Z}_p \mid y^2 \equiv x^3 + ax + b \mod p\right\} \cup \{\mathcal{O}\},$$

kjer $\mathcal{O}$ predstavlja točko v neskončnosti.

Na množici točk eliptične krivulje uvedemo operacijo seštevanja $+ : E \times E \to E$. Natančnih pravil za seštevanje tu ne bomo zapisali, saj so bila že predstavljena na predavanjih.

Opomba 2.3. Za izračun vsote dveh točk iz grupe E potrebujemo primerjanje, seštevanje, odštevanje, množenje in "deljenje" v $\mathbb{Z}_p$ (razen s točko $\mathcal{O}$). Z "deljenjem" mislimo na množenje z multiplikativnim inverzom v $\mathbb{Z}_p$. Seveda v primeru, ko namesto praštevila p vzamemo sestavljeno število n, inverz ne obstaja vedno. Ravno to nas bo pripeljalo do Lenstrinega algoritma.

V nadaljevanju bomo poleg vsote dveh točk iz grupe E, računali tudi večkratnike posameznih točk iz eliptične krivulje. Z kP označimo k-kratnik točke $P \in E$, kar je oznaka za več seštevanj iste točke:

$$kP = \underbrace{P + P + \cdots + P}_{k-\text{krat}}.$$

3. Algoritmi za faktorizacijo

3.1. Poskušanje

Najpreprostejši možen algoritem je poskušanje. Sestavljeno število n faktorziramo tako, da preprosto preizkusimo vse možne delitelje, začenši z 2, 3, ... Če nobeno izmed števil do $\lfloor\sqrt{n}\rfloor$ ne deli n, potem je n praštevilo. Postopek je bralcu gotovo dobro znan, zato natančnejši algoritem ne bo vključen.

Ena od možnih preprostih optimizacij algoritma je, da najprej preverimo, ali je število sodo. Če je sodo, potem je eden od faktorjev gotovo 2 (razen če je $n = 2$, a to je trivialen primer). V nasprotnem primeru zadostuje, da samo za vsa liha števila preverimo, ali delijo n.

V primeru, da želimo dobiti vse faktorje števila n, lahko algoritem izboljšamo tako, da ne končamo s postopkom takoj, ko najdemo enega izmed deliteljev i, temveč v tistem primeru zmanjšamo n na $\frac{n}{i}$, dokler i deli n, potem pa nadaljujemo s povečevanjem vrednosti i.

Ta preprost algoritem je uporaben za iskanje majhnih faktorjev, npr. za faktorje, manjše od 10^9. Če pa so vsi faktorji n veliki, postane uporaba tega algoritma nepraktična. Njegova časovna zahtevnost je $O(\sqrt{n})$, oziroma, če je n k-bitno število, $O(2^{\frac{k}{2}})$.

3.2. Fermatov algoritem

Za naslednji algoritem se omejimo samo na liha števila. To nam seveda ne pokvari ničesar, saj lahko na začetku preverimo, ali je n sodo. Če je, smo že našli enega izmed faktorjev. Če je n sestavljeno liho število, ga lahko zapišemo kot

$$n = k \cdot m, \quad k \leq m,$$

kjer sta tudi k in m lihi števili večji od 1. Označimo

$$x = \frac{1}{2}(k + m) \quad \text{in} \quad y = \frac{1}{2}(m - k). \tag{1}$$

Ker sta k in m lihi, sta x in y prav tako celi števili in velja:

$$\begin{aligned} x^2 - y^2 &= (x - y)(x + y) \\ &= \left(\frac{1}{2}(k + m) - \frac{1}{2}(m - k)\right)\left(\frac{1}{2}(k + m) + \frac{1}{2}(m - k)\right) \\ &= \frac{2}{2}k \cdot \frac{2}{2}m = n. \end{aligned}$$

To idejo razširimo. Poskusili bomo najti tako vrednost x, da bo veljalo $x^2 - n = y^2$. Od tu izhaja algoritem 1. Ta algoritem v nasprotju z algoritmom s poskušanjem, ki je našel najmanjši faktor, poišče največji faktor števila n, ki je še manjši ali enak $\sqrt{n}$.

Algoritem 1 Fermatov algoritem za faktorizacijo.

Vhod: $n \in \mathbb{N}$, lih, $n > 1$, ki ga želimo faktorizirati.
Izhod: En izmed deliteljev n.

1: $x = \lceil\sqrt{n}\rceil$
2: $y = x^2 - n$
3: **while not** y je popoln kvadrat **do**
4: $\quad y = y + 2x + 1$
5: $\quad x = x + 1$
6: **end while**
7: Vrnemo $x - \sqrt{y}$ (ali pa $x + \sqrt{y}$).

V primeru, ko je n praštevilo, Fermatova metoda porabi $O(\frac{n+1}{2} - \sqrt{n})$ preverjanj, ali je y popoln kvadrat. Tudi samo preverjanje, ali je y popol kvadrat ni trivialno, eden od možnih načino za preverbo je izračun korena, obstajajo pa tudi številne druge metode. V tem primeru je torej Fermatova metoda veliko počasnejša kot poskušanje. Je pa predstavljeni algoritem lahko precej hitrejši v primeru, ko lahko n zapišemo kot produkt faktorjev, ki sta zelo blizu $\sqrt{n}$. Naslednja trditev nam pove celo, da v primeru, ko je kateri od faktorjev n dovolj blizu $\sqrt{n}$ algoritem vrne rezultat že po enem samem koraku. Za dokaz glej Fischer [11].

Trditev 3.1. *Naj bo n liho sestavljeno naravno število. Naj velja $n = k \cdot m$ kjer je $|k - \sqrt{n}| < \sqrt[4]{n}$ ali $|m - \sqrt{n}| < \sqrt[4]{n}$. Potem Fermatov algoritem vrne enega izmed faktorjev po prvem koraku zanke* `while`.

Primer 3.2. Poiščimo faktor števila 4947851 s Fermatovim algoritmom:

$x = \lceil\sqrt{4947851}\rceil = 2225$

$y = 2225^2 - 4947851 = 2774,$ kar ni popoln kvadrat.

Vstopimo v `while` zanko:

$y = 2774 + 2 \cdot 2225 + 1 = 7225 = 85^2$ je popoln kvadrat. Vrnemo rezultat.

$x = 2225 + 1 = 2226$

Ker je y popoln kvadrat, smo našli faktorja, ki jih izračunamo takole:

$$k = 2226 - \sqrt{7225} = 2226 - 85 = 2141 \quad \text{in}$$
$$m = 2226 + \sqrt{7225} = 2226 + 85 = 2311.$$

Za konec naredimo še preizkus, $4947851 = 2141 \cdot 2311$. ♢

3.3. Faktorizacija z vzpenjanjem

Choudhury algoritem [4] uporabi idejo Fermatovega algoritma, doda pa ji še doda tehniko, ki se pogosto uporablja na področju umetne inteligence: *lokalno iskanje minimuma z vzpenjanjem* (angl. hill climbing technique).

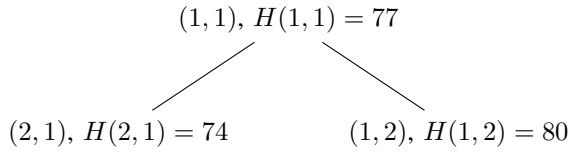
Obstaja veliko variacij algoritmov za vzpenjanje, v grobem pa je njihova ideja dokaj preprosta. Začnemo v nekem začetnem stanju. Na vsakem koraku izračunamo bližnja stanja. Če ocenimo, da je katero od bližnjih stanj boljše kot trenutno, se pomaknemo tja.

Določiti moramo stanje in najti še način, s katerim lahko za neko stanje ugotoviti, kako "blizu" faktoriziranemu številu smo. Definiramo hevristično funkcijo:

$$H(x, y) = n - (x^2 - y^2).$$

V primeru Fermatovega algoritma smo poskusili n zapisati kot $n = x^2 - y^2$. Če nam je to uspelo, smo lahko faktorizirali n. Torej v primeru, ko najdemo taki števili x in y, da je $H(x, y) = 0$, najdemo enega izmed faktorjev števila n. V vsakem koraku na stanju (x, y) nato preverimo dve sosednji stanji, stanje $(x + 1, y)$ in $(x, y+1)$. Premaknemo se v stanje z vrednostjo H bližje 0.

Primer 3.3. Faktorizirajmo število 77 z algoritmom 2. Začnemo v stanju $(1, 1)$, kjer je vrednost $H(1, 1) = 77-(1^2-1^2) = 77$. Izračunamo vrednosti H v sosednjih stanjih $(2, 1)$ in $(1, 2)$.



Algoritem 2 Faktorizacija z vzpenjanjem

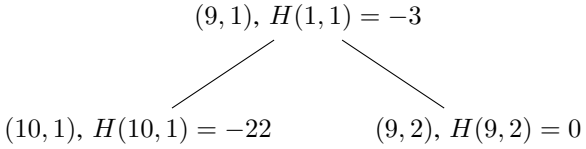
Vhod: $n \in \mathbb{N}$

Izhod: En od deliteljev n.

1: Nastavimo začetno stanje na $(x, y) = (1, 1)$
2: **if** $H(1, 1) = 1$ **then**
3: Končamo, v tem primeru je $n = 1$, torej je faktorizacija trivialna.
4: **end if**
5: **while** $H(x, y) \neq 0$ **do**
6: Premaknemo se v tisto sosednje stanje, ki ima vrednost funkcije H bližje 0.
7: **if** $|H(x + 1, y)| \leq |H(x, y + 1)|$ **and** $|H(x + 1, y)| \leq |H(x, y)|$ **then**
8: $(x, y) = (x + 1, y)$
9: **else if** $|H(x, y + 1)| \leq |H(x, y)|$ **then**
10: $(x, y) = (x, y + 1)$
11: **end if**
12: **end while**
13: Iz x in y izračunamo delitelja n, glej enačbo (1).

Ker je vrednost H v stanju $(2, 1)$ bližje 0 kot obe preostali vrednosti, se premaknemo v to stanje. Nadaljujemo tako, da izračunamo vrednost funkcije H v novih sosednih stanjih, $(3, 1)$ in $(2, 2)$. Postopek ponavljamo.

Po nekaj korakih pridemo do stanja $(9, 1)$, kjer ponovno izračunamo vrednosti v sosednjih stanjih $(10, 1)$ in $(9, 2)$:

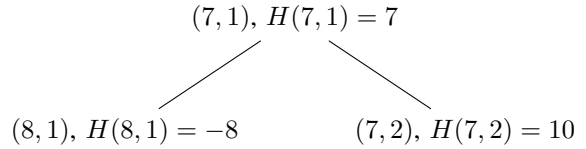


Pomaknemo se v stanje $(x, y) = (9, 2)$, ker je tam vrednost bližje 0. Ker je $H(9, 2) = 0$, z algoritmom zaključimo. Uporabimo enačbo (1) in izračunamo faktorja števila 77:

$$m = x + y = 9 + 2 = 11 \quad \text{in} \quad k = x - y = 9 - 2 = 7.$$

♢

Težava tega algoritma je, da pogosto ne najde nobenega faktorja. Če poskusimo faktorizirati število 55, na koncu pridemo do naslednjega stanja:



Iz stanja $(7, 1)$ se ne moremo premakniti na nobeno izmed možnih sosednjih stanj, saj je v obeh primerih absolutna vrednost $|H(x, y)|$ večja kot v trenutnem stanju. Algoritem obstane v stanju $(7, 1)$ in se nikoli

ne zaključi. Težava nastopi, ker faktorizacija celih števil nima "lepih", "zveznih" lastnosti. Iz primera, v katerem je $H(x, y)$ majhen, ne sledi, da smo blizu faktorizacije števila n.

Iz zgoraj naštetih razlogov je analiza algoritma precej zahtevna, zato so jo tudi avtorji članka [4] izpustili. Do morebitne pratkične uporabnosti podobnega algoritma bo potrebno vložiti še precej dela.

3.4. Pollardova metoda $p-1$

Britanski matematik John Pollard je prispeval dva znana algoritma za faktorizacijo, metodo ρ in metodo $p-1$, ki se sedaj imenujeta po njem. Ogledali si bomo slednjo, na kateri bazira tudi Lenstrin algoritem, ki ga bomo predstavili. Prepričali se bomo o njegovi pravilnosti in poskušali ugotoviti, v katerih primerih deluje hitro. S tem, zakaj algoritem deluje, se ne bomo ukvarjali, saj je bil algoritem že dokaj podrobno predstavljen na predavanjih. Postopek smo kljub temu zapisali v algoritmu 3, saj bo pomagal pri razumevanju Lenstrinega algoritma v nadaljevanju.

Algoritem 3 Pollardova metoda $p-1$

Vhod: $n \in \mathbb{N}$
Izhod: En od deliteljev n.

1: Izberemo naključno število a, $1 < a < n$.
2: **if** $\gcd(a, n) > 1$ **then**
3: Vrnemo $\gcd(a, n)$, saj je to eden od faktorjev števila n.
4: **end if**
5: **for** $r = 2, 3, 4, \ldots$ **do**
6: $a_r = a^{r!} \bmod n$
7: $d = \gcd(a_r - 1, n)$
8: **if** $d = n$ **then**
9: Vrnemo se na prvi korak in izberemo drugo število a.
10: **else if** $d \neq n$ **and** $d > 1$ **then**
11: Vrnemo d, saj je to eden od faktorjev števila n.
12: **end if**
13: **end for**

Pollardova $p-1$ metoda deluje hitro v primerih, ko je število $p-1$ sestavljeno iz majhnih praštevilskih faktorjev, t.j. *gladko praštevilo*. V primeru RSA, kjer je $n = pq$, torej ne želimo praštevil p in q, kjer bi bilo katero od števil $p-1$ in $q-1$ gladko. Težavi se izognemo tako, da izberemo *varni praštevili* p in q (angl. safe primes), tj. da sta tudi števili $\frac{p-1}{2}$ in $\frac{q-1}{2}$ praštevili.

Algoritem 3 lahko preprosto izboljšamo tako, da v vsaki iteraciji zanke ne računamo a_r od začetka, temveč si zapomnimo $a_{r-1} = a^{(r-1)!} \pmod n$ ter izračunamo samo $a_r = (a_{r-1})^r \pmod n$. Za potenciranje seveda uporabimo algoritem kvadriraj in zmnoži.

Primer 3.4. Poglejmo si, kako z algoritmom 3 poiščemo faktor števila 10001. Denimo, da naključno izberemo $a = 2$. Očitno je $\gcd(10001, 2) = 1$, tako da vstopimo v zanko. Izračunamo $a_2 = a^{2!} \bmod 1001 = 2^2 = 4$. Največji skupni delitelj

$$d = \gcd(4 - 1, 10001) = 1,$$

zato r povečamo in izračunamo $a_3 = (a_2)^3 \bmod 10001 = 4^3 = 64$ in

$$d = \gcd(64 - 1, 10001) = 1.$$

Postopek ponavljamo še za $r = 4, 5, \ldots$ V primeru $r = 6$ dobimo $a_6 = a^{6!} \bmod 10001 = 1169$. Največji skupni delitelj

$$d = \gcd(1169 - 1, 10001) = 73.$$

Tako smo našli enega od faktorjev števila 10001. Algoritem deluje, ker je za $p = 73$ število $p - 1$ sestavljeno iz majhnih faktorjev:

$$73 - 1 = 72 = 2^3 \cdot 3^2,$$

ki so vključeni v številu $6! = 2^4 \cdot 3^2 \cdot 5$. ♢

3.5. Lenstrin algoritem za faktorizacijo

Kot smo že omenili v poglavju 3.4, je Lenstrin algoritem zasnovan na Pollardovi $p-1$ metodi. Namesto v grupi $\mathbb{Z}_p$ delamo v neki grupi eliptične krivulje E. V grupi $\mathbb{Z}_p$ je operacija množenje, več množenj istega števila pa gledamo kot potenciranje. V grupi eliptične krivulje E pa je operacija seštevanje točk na krivulji. Več seštevanj iste točke pa je računanje večkratnika točke. Lenstrin algoritem nam pomaga faktorizirati število $n \in \mathbb{N}$, za katero velja $\gcd(n, 6) = 1$ in $n > 1$. Algoritem poteka takole:

1) Izberemo naključno eliptično krivuljo E in točko na njej $P \in E \setminus \{\mathcal{O}\}$. En od preprostejših načinov, kako to naredimo je, da najprej naključno izberemo $a \in \mathbb{Z}_n$ in $P = (x, y) \in \mathbb{Z}_n \times \mathbb{Z}_n$ ter izračunamo $b = y^2 - x^3 - ax$. Tako zagotovimo, da bo naključno izbrana točka gotovo na krivulji. V redkih primerih, ko je $\gcd(4a^3 + 27b^2, n) > 1$, E ni eliptična krivulja. Če je $\gcd(4a^3 + 27b^2, n) < n$, smo našli delitelja števila n, sicer pa krivuljo generiramo znova.
2) Izberemo $k \in \mathbb{N}$, ki ga deli veliko praštevil, npr. $k = B!$ ali $k = \mathrm{lcm}(2, 3, \ldots, B)$ za izbran B. Večji kot je B, več možnosti je, da bo algoritem uspešno faktoriziral n, vendar bo postopek trajal dlje.
3) Izračunamo točko kP. Pomagamo si z analogijo algoritma kvadriraj in zmnoži (angl. square and multiply), le da je to v tem primeru algoritem "podvoji in seštej", saj delamo v grupi, kjer je operacija seštevanje. Glej [5]. Pri vsakem posameznem seštevanju lahko pride do računanja

inverza v $\mathbb{Z}_n$, ki ga izračunamo z razširjenim Evklidovim algoritmom. Če je število n sestavljeno, tak inverz ne obstaja nujno. Pravzaprav inverz števila $m \in \mathbb{Z}_n$ obstaja natanko takrat, ko je $\gcd(m, n) = 1$. Torej v primeru, ko inverza ne moremo izračunati, dobimo enega od deliteljev števila n. V tem primeru algoritem končamo in vrnemo $\gcd(m, n)$.

4) Če nikjer med računanjem večkratnika kP nismo odkrili delitelja n lahko poskusimo ponovno z drugo naključno generirano krivuljo in točko, ali pa poskusimo s povečano vrednostjo števila B in novo krivuljo.

Primer 3.5. Uporabimo predstavljeno metodo za faktorizacijo števila 455839. Recimo, da je naša naključno izbrana krivulja $y^2 = x^3 + 5x - 5$ in točka $P = (1, 1)$. Denimo, da želimo izračunati $(10!)P$. Računamo:

$$\begin{aligned}(2!)P &= P + P = \ldots = (14, -53)\\ (3!)P &= (2P + 2P) + 2P = \ldots\\ &\vdots\end{aligned}$$

Za izračun večkratnika $(8!)P$ moramo izračunati inverz števila $599 \bmod 455839$. Ker pa je $\gcd(599, 455839) = 599$, inverz ne obstaja, smo pa tako našli enega od faktorjev števila 455839. $\diamondsuit$

Lenstrin algoritmem je verjetnostni algoritem, zato rešitve problema ne bomo našli vedno. Lahko se zgodi, da (zaradi slabe izbire začetnih vrednosti) rešitve sploh ne bomo našli, lahko pa zaradi neoptimalne izbire začetnih pogojev, rešitve ne bomo našli v času, ki ga imamo na razpolago. Zato je izbira parametra k v algoritmu zelo pomembna. Prav tako je pomembno dobro delovanje generatorja naključnih števil. S podrobno matematično analizo lahko ugotovimo, kako parametre izbrati čim bolj optimalno, vendar to presega obseg tega članka. Za več informacij glej Avsec [1]. V implementaciji sta zato B in k konstanti določeni s poskušanjem, neodvisni od vhodnih podatkov.

Naj bo p najmanjše praštevilo, ki deli število n, t.j. število, ki ga želimo faktorizirati. Potem je pričakovan čas izvajanja Lenstrinega algoritma, ki najde faktor p števila n, enak

$$O(e^{\sqrt{(2+O(1))\log p \log\log p}} \cdot (\log n)^2).$$

V najslabšem možnem primeru, ko nam uspe najti nek faktor števila n, je n produkt dveh praštevil istega reda velikosti. V tem primeru je pričakovan čas izvajanja

$$\begin{aligned}&O(e^{\sqrt{(2+O(1))\log n \log\log n}} \cdot (\log n)^2)\\ &\quad = O(n^{\sqrt{(2+O(1))\log\log n/\log n}}).\end{aligned}$$

Za več podatkov o časovni zahtevnosti glej [8].

3.6. Kvadratno sito

Kvadratno sito (angl. Quadratic Sieve) je algoritem, ki ga je iznašel Carl Pomerance leta 1981. Je trenutno v praksi drugi najhitrejši znani algoritem za faktorizacijo celih števil, takoj za številskim sitom (angl. Number Field Sieve) [10].

Naj bo n število, ki ga želimo faktorizirati. Denimo, da imamo taki števili x in y, da velja:

$$x^2 \equiv y^2 \mod n \quad \text{in} \quad x \not\equiv \pm y \mod n.$$

Potem je število $x^2 - y^2$ večkratnik števila n. Razliko kvadratov lahko razcepimo in dobimo:

$$x^2 - y^2 = (x - y)(x + y) = k \cdot n \quad \text{za nek } k \in \mathbb{Z}.$$

Torej ima število $x - y$ skupen faktor s številom n. Izračunamo ga kot $\gcd(x - y, n)$.

Če bi opustili pogoj $x \not\equiv \pm y \mod n$ je verjetnost, da faktor ne bo netrivialen, manjša, saj je lahko $x - y$ enako 0 ali pa večkratnik števila n.

Algoritem je sestavljen iz treh glavnih delov:

- generiranje baze za faktorizacijo,
- sejanja,
- generiranja matrike in reševanja sistema.

3.6.1. Generiranje baze za faktorizacijo

Najprej ustvarimo neko bazno množico praštevil $\mathcal{B} = \{p_1, p_2, \ldots, p_B\}$, npr. z uporabo Eratostenovega rešeta. Pogosto v bazo dodamo še faktor -1. Naj bo $p \in \mathcal{B}$. Če praštevilo p deli število n, smo našli faktor števila n in z algoritmom lahko končamo. V nadaljevanju predpostavimo, da je število n tuje vsem številom v bazi $\mathcal{B}$.

Izberemo si nek *sejalni interval* v okolici števila $\sqrt{n}$, npr. $[\lfloor\sqrt{n}\rfloor - M, \lfloor\sqrt{n}\rfloor + M]$. Za vsa cela števila x na tem intervalu definiramo

$$Q(x) = x^2 - n.$$

Poiskati želimo tiste vrednosti funkcije $Q(x)$, ki so gladke, t.j. tiste, ki jih lahko faktoriziramo z bazo $\mathcal{B}$.

Naj bo ponovno $p \in \mathcal{B}$. Če p deli $Q(x)$, to pomeni, da je $x^2 - n \equiv 0 \pmod p$ oziroma

$$x^2 \equiv n \pmod p.$$

Ravnokar smo ugotovili, da je n kvadratni ostanek števila x. Po Eulerjevem kriteriju (glej Stinson in Paterson [5]) se omejimo samo na tiste $p \in \mathcal{B}$, za katere velja:

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

Dobimo novo bazo praštevil $\mathcal{B}'$, ki jo imenujemo *baza faktorjev*. Večja, kot je baza faktorjev, več možnosti imamo, da nam uspe faktorizirati število n. Ni pa vedno bolje imeti čim večje baze. Poleg samega generiranja baze in sejanja (v nadaljevanju) bomo reševali še sistem linearnih enačb velikosti $|\mathcal{B}'|\times|\mathcal{B}'|$, kar je lahko za velike $|\mathcal{B}|$ počasno.

3.6.2. Sejanje

V tem koraku začnemo z izločanjem elementov x iz sejalnega intervala. Obdržimo samo tista števila x, za katera lahko $Q(x)$ zapišemo kot produkt baznih praštevil.

Za večjo učinkovitost v praksi tega ne delamo direktno na opisan način. Poglejmo si naslednjo enačbo:

$$\begin{aligned} Q(x+p) &= (x+p)^2 - n = x^2 + 2px + p^2 - n \\ &\equiv x^2 - n \equiv Q(x) \pmod{p}. \end{aligned}$$

Od tod takoj opazimo naslednjo lastnost: Če p deli $Q(x)$, potem p deli tudi $Q(x+p)$. Zato sejanja ne izvajamo za vsak element sejalnega intervala posebej, temveč za vsako praštevilo p v bazi faktorjev rešimo enačbo

$$Q(x) \equiv 0 \pmod{p} \quad \text{oziroma} \quad x^2 \equiv n \pmod{p}$$

za en x, ter potem hkrati faktoriziramo še vsa števila $x + kp, k \in \mathbb{Z}$. Tudi ta način razbitja števil na produkt baznih praštevil je dokaj preprosto paralelizirati. V tem primeru moramo izračunati kvadratni koren števila n po modulu p. To lahko naredimo s Tonelli-Shanksovim algoritmom. Glej [12] za več podrobnosti. Paziti moramo, da upoštevamo obe možni rešitvi, x_{1p} in $x_{2p} = p - x_{1p}$.

3.6.3. Generiranje matrike in reševanje sistema

V nadaljevanju bi radi našli taka števila $x_1, \ldots, x_k$ izmed preostalih vrednosti v sejalnem intervalu, da bo produkt $Q(x_1)Q(x_2)\cdots Q(x_k)$ popoln kvadrat. Recimo, da nam to uspe in je $y^2 = Q(x_1)Q(x_2)\cdots Q(x_k)$ za neko število y. Potem je

$$\begin{aligned} y^2 &= (x_1^2 - n)(x_2^2 - n)\cdots(x_k^2 - n) \\ &\equiv x_1^2 x_2^2 \cdots x_k^2 \pmod{n}. \end{aligned}$$

Od tu naprej postopamo tako, kot že omenjeno na začetku tega poglavja. Izračunamo lahko $\gcd(x_1 x_2 \cdots x_k - y, n)$.

Že v prejšnjem koraku smo vrednosti $Q(x)$ faktorizirali glede na bazo $\mathcal{B}'$. Na bazo gledamo kot na bazo vektorskega prostora nad obsegom $\mathbb{Z}_2$ in vrednostim $Q(x)$ priredimo vektorje. Dobljene vektorje zložimo v vrstice matrike A.

Primer 3.6. Naj bo $\mathcal{B}' = \{-1, 2, 3, 13, 17, 19\}$ faktorizacijska baza in $Q(x) = 2 \cdot 3 \cdot 13^2 \cdot 19$. Potem je vektor, ki ga priredimo vrednosti $Q(x)$ enak

$$(0,1,1,2,0,1) \equiv (0,1,1,0,0,1) \pmod{2}. \qquad \diamondsuit$$

Opazimo, da bo produkt $Q(x_1)Q(x_2)\cdots Q(x_k)$ popoln kvadrat natanko tedaj, ko bo vsota vektorjev, prirejenih vrednostim $Q(x_i)$ enaka 0 (ker smo v prostoru nad obsegom $\mathbb{Z}_2$). Označimo z $\vec{a}_i$ i-to vrstico matrike A, ki je prirejena vrednosti $Q(x_i)$. Naš problem lahko prevedemo na iskanje takih koeficientov $\alpha_i \in \mathbb{Z}_2$, da bo

$$\alpha_1\vec{a_1} + \alpha_2\vec{a_2} + \cdots + \alpha_k\vec{a_k} \equiv \vec{0} \mod 2,$$

oziroma reševanje sistema

$$A\vec{\alpha} \equiv \vec{0} \mod 2.$$

Z linearno algebro poiščemo nek neničelni vektor iz jedra matrike A. Najpreprostejši način za to je, da uporabimo Gaussovo eliminacijo. Ko najdemo tak vektor, lahko poizkusimo izračunati nek faktor števila n. Če nam faktorizacija ne uspe (ker smo dobili bodisi 0 bodisi nek večkratnik n), poskusimo z nekim drugim linearno neodvisnim vektorjem iz jedra matrike A^T. Da si zagotovimo dovolj neodvisnih vektorjev v jedru, zahtevamo, da ima matrika A več vrstic kot stolpcev, npr. zahtevamo, da nam po sejanju ostane vsaj $|\mathcal{B}'| + 10$ vrednosti sejalnega intervala.

3.6.4. Časovna zahtevnost

Časovna zahtevnost algoritma je odvisna od velikosti baze faktorjev in velikosti sejalnega intervala. Izkaže se, da je optimalna velikost baze faktorjev približno

$$|\mathcal{B}'| = \left(e^{\sqrt{\log(n)\log(\log n)}}\right)^{\frac{\sqrt{2}}{4}}$$

in širina intervala približno

$$2M = |\mathcal{B}'|^3 = \left(e^{\sqrt{\log(n)\log(\log n)}}\right)^{\frac{3\sqrt{2}}{4}}.$$

Glej Bogataj [3]. Ko upoštevamo še časovno zahtevnost Gaussove eliminacije, dobimo skupno časovno zahtevnost (Yan [8])

$$O(n^{(1+O(1))\sqrt{\log(\log n)/\log n}}),$$

kar je enako kot pri Lenstrinem algoritmu, a je v praksi kvadratno sito hitrejše [10].

4. IMPLEMENTACIJA ALGORITMOV

Zgoraj opisane algoritme smo tudi implementirali. Izbran je bil programski jezik Python. Za merjenje hitrosti implementiranega algoritma Python še zdaleč ni dobra izbira. Je pa zaradi podpore dela s poljubno velikimi števili celotna implementacija v jeziku Python veliko bolj preprosta. Glavni cilj projekta je razumevanje delovanja algoritmov in ne njihova učinkovita implementacija.

Učinkovitost implementiranih algoritmov je težko primerjati med sabo, saj je njihov čas izvajanja odvisen od strukture posameznega števila. Poleg tega je težko primerjati deterministične algoritme, kot je poskušanje, z verjetnostnimi algoritmi, ki včasih vrnejo pravilen rezultat, včasih pa ne. Zaradi naštetih razlogov sem se odločil, da ne bom delal primerjave med implementiranimi algoritmi.

5. ZAKLJUČEK

Namen članka je bil spoznati in implementirati nekatere bolj znane algoritme za faktorizacijo števil. Kljub temu, da je problem faktorizacije eden starejših matematičnih problemov, je aktualen še danes, saj na njem temelji varnost RSA algoritma. S trenutno znanimi algoritmi je RSA trenutno še varen algoritem, seveda pa se lahko v prihodnosti ob morebitnem odkritju novih algoritmov to povsem spremeni.

APPENDIX

V nadaljevanju je vključena koda implementacije predstavljenih algoritmov.

trial_division.py

```python
from math import isqrt


def trial_division(n):
    """
    Performs trial division factoring algorithm.
    Returns first factor found.
    If there aren't any nontrivial factors (i.e. n is a prime or 1), None is returned.
    """
    if n == 1 or n == 2:
        return None
    if n % 2 == 0:
        return 2

    i = 3
    # Calculate integer square root of n
    # (Python does this correctly, so we do not loose digits after 16th decimal place)
    sq = isqrt(n)
    while i <= sq:
        if n % i == 0:
            return i
        i += 2

    # No factors found, n is a prime.
    return None


# Example numbers to factorize
to_factor = [
    49,
    15,
    101 * 103,  # products of two primes
    1009 * 1013,
    10007 * 10009,
    2 ** 50]

for n in to_factor:
    print(f'Factoring {n}')
    factor = trial_division(n)
    print(factor)
```

fermats_factorization.py

```python
from math import isqrt


def is_perfect_square(n):
    """
    Checks if given n is a perfect square.
    """
    return n == isqrt(n) ** 2


def fermats_factorization(n):
    """
    Performs Fermat's factorization algorithm on number n.
    Returns smaller one of the found factors.
    If there aren't any nontrivial factors (i.e. n is a prime or 1), None is returned.
    """
    if n == 1 or n == 2:
        return None
    if n % 2 == 0:
        # Algorithm does not work for even numbers, but we know how to factorize them.
        return 2
```

```
    # Calculate ceil(sqrt(n))
    # See isqrt docs on why formula below is correct.
    x = 1 + isqrt(n - 1)
    y = x ** 2 - n

    while not is_perfect_square(y):
        y += 2 * x + 1
        x += 1

    return x - isqrt(y)


# Example numbers to factorize
to_factor = [
    49,
    15,
    # all primer pairs below are close together, Fermat's algorithm should be very fast.
    101 * 103,  # products of two primes
    1009 * 1013,
    10007 * 10009,
    1000000009 * 1000000007,
    10000000019 * 10000000033,
    100000000003 * 100000000019,
    1000000000039 * 1000000000061,
    29996224275833 * 29996224275821,
    2 ** 50]

for n in to_factor:
    print(f'Factoring {n}')
    factor = fermats_factorization(n)
    print(factor)
```

hill_climbing.py

```
def get_H(n):
    """
    Returns heuristic function H algorithm is optimizing for.
    """
    return lambda x, y: n - (x ** 2 - y ** 2)


def hill_climbing(n):
    """
    Performs Hill-factor algorithm on number n.
    Returns one of the found factors or None if no factors found.
    """
    H = get_H(n)
    x, y = 1, 1

    if H(x, y) == 0:
        return x - y

    current = abs(H(x, y))
    while current != 0:
        left = abs(H(x + 1, y))
        right = abs(H(x, y + 1))
        if current < left and current < right:
            # Current node is smallest, stuck in infinite cycle
            return None
        # Go to state with smaller value
        if left <= right:
            x += 1
            current = left
        else:
            y += 1
            current = right
```

```
    # Return one of the factors
    return x - y


# Example numbers to factorize
to_factor = [77, 15, 101 * 103, 1009 * 1013, 10007 * 10009]

for n in to_factor:
    print(f'Factoring {n}')
    factor = hill_climbing(n)
    print(factor)
```

pollard_p_minus_1.py

```
from random import randint
from math import gcd


def pollard_p_minus_1(n, m, tries=100):
    """
    Tries to factor n using Pollard's p - 1 method.
    Parameter m is max value of r in algorithm where we are calculating a^(r!).
    Returns one of the factors if successful, otherwise None.
    """
    for _ in range(tries):
        a = randint(2, n - 1)

        for r in range(2, m + 1):
            # Calculate a^(r!)
            a = pow(a, r, n)
            d = gcd(a - 1, n)

            if d == n:
                break
            elif d > 1:
                # We found a factor
                return d

    return None


# Example numbers to factorize
to_factor = [
    49,
    15,
    101 * 103,  # products of two primes
    1009 * 1013,
    10007 * 10009,
    1000000009 * 1000000007,
    2 ** 50]

m = 100000

for n in to_factor:
    print(f'Factoring {n}')
    factor = pollard_p_minus_1(n, 1000)
    print(factor)
```

ecm.py

```
import random


def extended_gcd(a, b):
    """
    Extended Euclidean algorithm. Given a and b calculates gcd(a, b), s and t where
    a * s + b * t = gcd(a, b)
    """
    # Follow a table for egcd calculation.
```

```
    s0, t0 = 1, 0
    s1, t1 = 0, 1

    while b != 0:
        # Calculate quotient
        q = a // b

        # Subtract q * last value.
        s0, s1 = s1, s0 - q * s1
        t0, t1 = t1, t0 - q * t1

        # Next step
        a, b = b, a % b

    return a, s0, t0


def inverse(a, n):
    """
    Calculates inverse of a mod n.
    If this is not possible, we found a factor of n.
    Returns tuple (inverse, factor), exactly one of them should not be None.
    """
    d, s, _ = extended_gcd(a, n)

    if d != 1:
        # Cannot calculate inverse - found a factor of n
        return None, d

    # Return inverse
    return s, None


def random_elliptic_curve(n):
    """
    Generates random elliptic curve and point on it. All in Z_n.
    Returns tuple
    (curve, point)
    Returned curve is represented as a tuple (n, a, b) which means
    y^2 = x^3 + ax + b
    """
    # Generate random point.
    x = random.randint(0, n - 1)
    y = random.randint(0, n - 1)

    # Generate random curve which contains this point.
    # Choose a randomly and calculate b from equation
    a = random.randint(0, n - 1)
    b = (y ** 2 - x ** 3 - a * x) % n

    return ((n, a, b), (x, y))


def add_points(curve, P, Q):
    """
    Addition in elliptic curves.
    None represents point infinity.
    Returns tuple (R, factor) where R = P + Q and factor is factor
    if we have factorized n (usually factor will be None).
    """
    # Addition with infinity
    if P is None:
        return Q, None
    if Q is None:
        return P, None
```

```
    # Unpack
    n, a, b = curve
    px, py = P
    qx, qy = Q

    # Point to return
    R = None

    if px != qx:
        # First option: x1 != x2
        numerator = (qy - py) % n
        denominator = (qx - px) % n

        denominator, factor = inverse(denominator, n)
        if factor is not None:
            # We found a factor
            return None, factor

        fraction = (numerator * denominator) % n

        x = (fraction ** 2 - px - qx) % n
        y = (-py + fraction * (px - x)) % n
        R = (x, y)
    elif P == Q and py != 0:
        # Second option: both points are the same and y != 0
        numerator = (3 * px ** 2 + a) % n
        denominator = (2 * py) % n

        denominator, factor = inverse(denominator, n)
        if factor is not None:
            # We found a factor
            return None, factor

        fraction = (numerator * denominator) % n

        x = (fraction ** 2 - 2 * px) % n
        y = (-py + fraction * (px - x)) % n
        R = (x, y)
    elif px == qx and (py ** 2) % n == (qy ** 2) % n:
        # Third option: points are symetric over x axis
        # Result is infinity, no factor
        R = None
    else:
        raise ValueError(
            f'Unhandled case. Should not happen. Points P={P} and Q={Q} on curve {curve}.')

    # We haven't found the factor.
    return R, None


def multiply_point(curve, P, m):
    """
    Adds point P together m-times.
    mP = P + P + ... + P
    Returns tuple (result, factor) if we found a factor.
    """
    # Start with infinity and add point to this.
    R = None
    # Algorithm almost identical to square and multiply, except with addition
    while m > 0:
        if m % 2 != 0:
            # if there is 1 in binary add current factor
            R, factor = add_points(curve, R, P)
            if factor is not None:
                # We found a factor
                return None, factor
```

```
        # Square part of square and multiply algorithm
        P, factor = add_points(curve, P, P)
        m >>= 1

        if factor is not None:
            # We found a factor
            return None, factor

    # No factor found
    return R, None


def ecm(n, k, tries=100):
    """
    Tries to factorize n using Lenstra's Elliptic Curve Method.
    k should be a large number consisting of small primes (for example B!).
    Returns factor of n if successful, otherwise None.
    """
    for i in range(tries):
        curve, point = random_elliptic_curve(n)
        _, factor = multiply_point(curve, point, k)
        if factor is not None:
            return factor

    return None


def prod(l):
    """
    Calculate product of elements in a list.
    """
    r = 1
    for i in l:
        r *= i
    return r


print('Generating product k = B!')
B = 1000
k = prod(range(2, B + 1))

# Example numbers to factorize
to_factor = [
    49,
    15,
    101 * 103,  # products of two primes
    1009 * 1013,
    10007 * 10009,
    1000000009 * 1000000007,
    10000000019 * 10000000033,
    100000000003 * 100000000019,
    1000000000039 * 1000000000061,
    29996224275833 * 29996224275821,
    2 ** 50]

for n in to_factor:
    print(f'Factoring {n}')
    factor = ecm(n, k)
    print(factor)
```

quadratic_sieve.py

```
from math import isqrt, gcd, prod


def generate_primes(n):
    """
```

```
    Generate list of primes less or equal than n.
    Using Sieve of Eratostenes.
    """
    # No primes for n less than 2
    if n < 2:
        return []

    # is_prime[i] will be True if i is prime.
    # Ignore index 0 (used for easier indexation)
    is_prime = [True] * (n + 1)
    is_prime[0] = False
    is_prime[1] = False  # 1 is not a prime

    # Set all even numbers to False (except 2)
    is_prime[4:: 2] = [False] * ((len(is_prime) - 2) // 2)

    for i in range(3, isqrt(n) + 1, 2):
        if is_prime[i]:
            # We found a next prime.
            # Set all multiples of i to False
            # use Python's slices as they are much faster than for loop
            is_prime[2 * i:: i] = [False] * ((len(is_prime) - i - 1) // i)

    # Return indices = primes
    return [i for i, prime in enumerate(is_prime) if prime]


def generate_base(n, m):
    """
    Generate base for factoring number n.
    Base will contain primes up to m.
    """
    # Generate primes up to B
    primes = generate_primes(m)
    # Filter primes to keep only ones with quadratic residues
    return [p for p in primes if pow(n, (p - 1) // 2, p) == 1]


def factor(n, base):
    """
    Factor number n using base.
    Expects -1 to be at the start of the base.
    """
    factors = [0 for i in range(len(base))]
    # -1 is special factor (needs to be added to matrix)
    if n < 0:
        factors[0] += 1
    for i, p in enumerate(base):
        if p == -1:
            continue
        while n % p == 0:
            factors[i] += 1
            n //= p
    return factors


def transpose(M):
    """
    Transpose matrix represented as a 2D Python list.
    """
    return [[row[i] for row in M] for i in range(len(M[0]))]


def is_smooth(n, base):
    """
    Checks if number can be factorized using base.
```

```
    """
    if n == 0:
        return False
    for factor in base:
        while n % factor == 0:
            n //= factor
    return n == 1


def tonelli_shanks(n, p):
    """
    Algorithm for solving equation x ** 2 = n (mod p),
    modular square root. Works only for odd primes.
    https://en.wikipedia.org/wiki/Tonelli-Shanks_algorithm

    Returns x, such that x ** 2 = n (mod p) if such x exists.
    Otherwise returns None
    """
    # Works only for even numbers
    assert p % 2 == 1

    # We want that this is quadratic residue,
    # otherwise solution won't exist
    if n ** ((p - 1) // 2) % p != 1:
        return None

    # Factor p - 1, following algorithm from Wikipedia
    # p - 1 = Q * 2 ** S
    Q = p - 1
    S = 0
    while Q % 2 == 0:
        Q //= 2
        S += 1

    # Search for such z that Jacobi (z/p) == -1
    # (quadratic non-residue)
    for z in range(2, p):
        # Since p is prime we have a simple formula.
        if z ** ((p - 1) // 2) % p == p - 1:
            break

    # Formulas copied from Wikipedia
    M = S
    c = pow(z, Q, p)
    t = pow(n, Q, p)
    R = pow(n, ((Q + 1) // 2), p)

    while t != 1:
        for i in range(1, M):
            if t ** (2 ** i) % p == 1:
                break
        b = c ** (2 ** (M - i - 1)) % p
        R = (R * b) % p
        t = (t * b * b) % p
        c = (b * b) % p
        M = i
    # This is one solution. Second solution is p - R.
    return R


def generate_smooth(n, base, interval_width):
    """
    Generate smooth values near sqrt(n).
    """
    sq = isqrt(n)
    # Numbers Q from the algorithm.
```

```
    sieve = [x**2 - n for x in range(sq, sq + interval_width)]
    sieve_copy = sieve[:]

    # Tonelli-Shanks' algorithm doesn't work for even primes (for 2)
    if base[0] == 2:
        # Divide with 2 as long as we can,
        for i in range(len(sieve)):
            while sieve[i] % 2 == 0:
                sieve[i] //= 2

    # All other primes
    for p in base[1:]:
        # Calculate square root of n mod p,
        r1 = tonelli_shanks(n, p)
        # We have two solutions of x**2 = n (mod p)
        res = [r1, (p - r1) % n]

        for r in res:
            # Divide with p as long as we can.
            # Here we can skip a lot of steps, only need to check every p-th number.
            for i in range((r - sq) % p, len(sieve), p):
                while sieve[i] % p == 0:
                    sieve[i] //= p

    xs = []
    smooth = []
    for i in range(len(sieve)):
        # If we found enough numbers
        if len(smooth) >= len(base) + 10:
            break
        if sieve[i] == 1:
            # We found a smooth number, get original (not divided number)
            smooth.append(sieve_copy[i])
            # Original x value (before squaring)
            xs.append(i + sq)

    return xs, smooth


def build_matrix(smooth, base):
    """
    Builds matrix of smooth numbers according to base.
    Everything is done mod 2 as we are solving for squares.
    """
    M = []
    # Add -1 to base
    base = [-1, *base]

    for n in smooth:
        # Get vector for each smooth number
        v = factor(n, base)
        # Calculate everything mod 2
        v = [i % 2 for i in v]
        M.append(v)
    return transpose(M)


def gaussian_elimination(M):
    """
    Performs gaussian elimination on the matrix M.
    """
    marks = [False for i in range(len(M[0]))]

    for i in range(len(M)):
        # We are in i-th row.
        row = M[i]
```

```
        for j, val in enumerate(row):
            # val = M_ij
            if val == 1:
                marks[j] = True

                # Destroy other ones in this column
                for k in range(0, len(M)):
                    if k == i:
                        # Skip current row
                        continue
                    # Add row (adding is same as subtraciton is Z_2)
                    if M[k][j] == 1:
                        for i in range(len(M[k])):
                            M[k][i] = (M[k][i] + row[i]) % 2
                break

    # Transpose for easier access (2D arrays ...)
    M = transpose(M)
    # Get potential solutions
    solution = []
    for i in range(len(M)):
        if marks[i] == False:
            free_row = [M[i], i]
            solution.append(free_row)

    if len(solution) == 0:
        return None, None, None

    return solution, marks, M


def solve_row(solution, marks, M, k=0):
    """
    Second part of gauss, solve for rows.
    Get one of possible solutions.
    """
    s, indices = [], []
    free_row = solution[k][0]
    for i in range(len(free_row)):
        if free_row[i] == 1:
            indices.append(i)
    # Rows with 1 in the same column will be dependent
    for r in range(len(M)):
        for i in indices:
            if M[r][i] == 1 and marks[r]:
                s.append(r)
                break

    s.append(solution[k][1])
    return s


def solve(solution, smooth, xs, n):
    """
    Try to calculate factor of n using one possible solution.
    """
    xs = [xs[i] for i in solution]
    solution = [smooth[i] for i in solution]

    # Calculate product of all solutions
    a_squared = prod(solution)
    a = isqrt(a_squared)

    b = prod(xs)

    return gcd(b - a, n)
```

```
def quadratic_sieve(n, m):
    """
    Tries to factor n using quadratic sieve method.
    Primes up to m are used in base.
    """
    base = generate_base(n, m)

    B = len(base)
    print(f'Generated a base of size {B}.')

    xs, smooth = generate_smooth(n, base, 100000)
    print(f'Found {len(smooth)} smooth numbers.')

    print("Gaussian elimination")
    matrix = build_matrix(smooth, base)
    solution, marks, matrix = gaussian_elimination(matrix)
    s = solve_row(solution, marks, matrix)

    fact = solve(s, smooth, xs, n)

    for k in range(1, len(solution)):
        if fact == 1 or fact == n:
            # Try different solution vector
            s = solve_row(solution, marks, matrix, k)
            fact = solve(s, smooth, xs, n)
        else:
            # We found nontrivial factor
            return fact
    return None


m = 10000
# Example numbers to factorize
to_factor = [
    1009 * 1013,
    10007 * 10009,
    1000000009 * 1000000007,
    10000000019 * 10000000033,
    100000000003 * 100000000019,
    1000000000039 * 1000000000061,
    29996224275833 * 29996224275821]

for n in to_factor:
    print(f'Factoring {n}')
    print(quadratic_sieve(n, m))
```

REFERENCES

[1] M. Avsec: *Kubične krivulje v kriptografiji*, diplomsko delo, Univerza v Ljubljani, FMF 2020. https://repozitorij.uni-lj.si/IzpisGradiva.php?lang=slv&id=116144.

[2] C. Barnes: *Integer factorization algorithms*. Department of Physics, Oregon State University, 2004.

[3] P. Bogataj *Faktorizacija naravnih števil*, diplomsko delo. Univerza v Ljubljani, FRI, 2011.

[4] B. Choudhury in S. Neog. *Finding Factors of a Number Using Steepest Ascent Hill Climbing*. International Journal of Electronics and Applied Research **2** (2015), str. 29–34.

[5] D. R. Stinson, M. Paterson, *Cryptography - Theory and Practice*, CRC Press, 4. izdaja, 2018

[6] E. Landquist *The Quadratic Sieve Factoring Algorithm* Math **488** (2001), str. 1–11.

[7] P. L. Montgomery: *A survey of modern integer factorization algorithms*. CWI quarterly, 1994, str. 337 – 366.

[8] S. Y. Yan: *Primality testing and integer factorization in public-key cryptography*, Springer 2009.

[9] *Fermat's Factorization Method*, v: Wikipedia, The Free Encyclopedia, [dostopno 3. 1. 2021], en.wikipedia.org/wiki/Fermat%27s_factorization_method.

[10] *Quadratic Sieve*, v: Wikipedia, The Free Encyclopedia, [dostopno 12. 2. 2021], https://en.wikipedia.org/wiki/Quadratic_sieve.

[11] *Fermat's factorization in one step* v: Mathematics Stack Exchange, [dostopno 4. 1. 2021], math.stackexchange.com/questions/3964455/fermat-factorisation-in-one-step.

[12] *Tonelli–Shanks algorithm* v: Wikipedia, The Free Encyclopedia, [dostopno 13. 2. 2021], https://en.wikipedia.org/wiki/Tonelli%E2%80%93Shanks_algorithm.