Algebarski algoritmi

Uvek kada izvršavamo neku algebarsku operaciju, kao što je recimo množenje dva broja ili njihovo stepenovanje, mi u stvari izvršavamo neki algoritam. Mi te operacije koristimo kao gradivne elemente u razvijanju složenijih algoritama i često ne zalazimo dublje u analizu njihove složenosti. Međutim, i sami algoritmi sabiranja, oduzimanja, množenja, deljenja i stepenovanja brojeva (posebno ako su brojevi dati nizovima svojih cifara) predstavljaju važne algebarske algoritme. U algebarske algoritme spadaju i mnogi algoritmi sa kojima smo se već susreli kao što su brzo stepenovanje broja, izračunavanje vrednosti broja na osnovu datih cifara ili, nasuprot tome, određivanje cifara broja na osnovu njegove vrednosti, računanje najvećeg zajedničkog delioca dva data broja, zatim razni algoritmi nad polinomima kao što su izračunavanje vrednosti polinoma i množenje polinoma. U ovom poglavlju bavićemo se algebarskim algoritmima sa kojima se do sada nismo susreli. Mnogi od njih igraju važnu ulogu u oblasti kriptografije, ali i u drugim oblastima, poput algebarske teorije brojeva i kvantnog računarstva.

Modularna aritmetika

Kažemo da je broj r ostatak pri deljenju broja x brojem y i pišemo $x \bmod y = r$ ako i samo ako postoji broj q takav da je $x = q \cdot y + r$ i $0 \leq r < y$. Broj r je ovim uslovom jedinstveno određen. Pored toga što sa mod označavamo binarnu operaciju, mod se koristi i kao oznaka binarne relacije u skupu celih brojeva. Naime, pisaćemo $a \equiv b \pmod{m}$ ako $m|a - b$. Ovo odgovara tome da a i b daju isti ostatak pri deljenju sa m. Na primer, $12 \equiv 2 \pmod{5}$ jer $5|(12 - 2)$.

Relaciju mod koristimo u nekim svakodnevnim situacijama, a da toga često nismo ni svesni. Jedan od takvih primera je rad sa vremenom. Naime, za vreme koje je 15 časova nakon 11 sati reći ćemo da je 2 sata (što odgovara tome da je $11 + 15 \equiv 2 \pmod{24}$). Slično važi i za dane u nedelji koje računamo po modulu 7: ako je danas četvrtak (četvrti dan po redu u sedmici), za šest dana biće sreda (treći po redu dan u sedmici), jer je $4 + 6 \equiv 3 \pmod{7}$. Analogno se računa i mesec ili redni broj nedelje u godini.

Modularna aritmetika ima puno praktičnih primena: koristi se za izračunavanje kontrolnih suma za međunarodne standardne identifikatore knjiga (ISBN brojeve), međunarodne brojeve bankovnih računa (IBAN), kao i jedinstvene identifikatore hemijskih jedinjenja (CAS registarski broj). Modularna aritmetika je i u osnovi savremenih kriptografskih sistema.

Brojevi se u računarima predstavljaju po modulu 2^k. Recimo, u jeziku C++ brojevi tipa `unsigned int` predstavljaju se po modulu 2^{32}. Na primer, u narednom

kodu rezultat kvadriranja broja 123456789 biće vrednost $123456789^2 \bmod 2^{32} = 2537071545$.

```
unsigned int x = 123456789;
cout << x * x << endl;
```

Ako je $a \equiv b \pmod{m}$, onda je $a \bmod m = b \bmod m$.

Ako je $a \equiv b \pmod{m}$ i $c \equiv d \pmod{m}$, onda je $a \pm c \equiv b \pm d \pmod{m}$ i $a \cdot c \equiv b \cdot d \pmod{m}$; na primer,

$$\begin{array}{ccc} 12, 14 & \xrightarrow{\text{mod } 5} & 2, 4 \\ + \downarrow & & \downarrow + \\ 26 & \xrightarrow{\text{mod } 5} & 1 \end{array}$$

Drugim rečima, relacija $\bmod$ je *saglasna* sa operacijama sabiranja, oduzimanja i množenja.

Kao posledica ovog tvrđenja važe sledeća tvrđenja za vrednost zbira ili proizvoda brojeva po modulu m:

$$(a + b) \bmod m = (a \bmod m \ + \ b \bmod m) \bmod m$$

$$(a \cdot b) \bmod m = (a \bmod m \ \cdot \ b \bmod m) \bmod m$$

Dokažimo drugu relaciju (prva je jednostavnija za dokazivanje). Pretpostavimo da je $a = q_a \cdot m + r_a$ i $b = q_b \cdot m + r_b$ za $0 \leq r_a, r_b < m$. Tada važi da je:

$$a \cdot b = (q_a \cdot m + r_a) \cdot (q_b \cdot m + r_b) = (q_a \cdot q_b \cdot m + q_a \cdot r_b + r_a \cdot q_b)m + r_a \cdot r_b.$$

Ako važi da je:

$$r_a \cdot r_b = q \cdot m + r, \ 0 \leq r < m,$$

tada je:

$$a \cdot b = (q_a \cdot q_b \cdot m + q_a \cdot r_b + r_a \cdot q_b + q)m + r, \ 0 \leq r < m$$

pa je:

$$(a \cdot b) \bmod m = r.$$

Važi, takođe, da je:

$$(a \bmod m \ \cdot \ b \bmod m) \bmod m = (r_a \cdot r_b) \bmod m = r,$$

čime je tvrđenje dokazano.

Razmotrimo problem određivanja razlike nenegativnih brojeva b i a po modulu m. Na primer, potrebno je odrediti vrednost $(b - a) \bmod m$ za $a, b \geq 0$. Bez smanjenja opštosti može se pretpostaviti da su brojevi a i b manji od m. U slučaju kada je $b < a$ mi bismo želeli da kao vrednost ovog izraza dobijemo nenegativnu vrednost. Napomenimo da ne postoji saglasnost između različitih programskih

jezika u računanju vrednosti `a % m` kada je vrednost broja `a` negativna. Naime, u jeziku C++ se za vrednost ostatka pri deljenju može dobiti negativan broj: na primer, vrednost izraza `(2 - 7) % 3` je -2, dok se u jeziku Python kao rezultat istog ovog izraza dobija pozitivan broj 1. Mi bismo želeli da kao rezultat dobijemo $r \geq 0$. Umesto da vršimo analizu slučajeva, rešenje je moguće dobiti izračunavanjem vrednosti izraza

$$(b - a + m) \bmod m.$$

Zaista, ako je $b \geq a$ i $0 \leq a, b < m$, vrednost izraza $b - a + m$ biće veća ili jednaka m i njen ostatak pri deljenju sa m biće jednak vrednosti $b - a$ (traženje ostatka će praktično poništiti prvobitno dodavanje vrednosti m). Sa druge strane, ako je $b < a$ i $0 \leq a, b < m$ tada će $b - a$ biti negativan ceo broj iz intervala $[-m+1, -1]$, pa će se dodavanjem vrednosti m dobiti ceo broj iz intervala $[1, m-1]$. Vrednost ostatka u ovom slučaju bi'e jednaka samom argumentu, odnosno kao vrednost ostatka pri deljenju sa m dobiće se vrednost $b - a + m$, što smo i želeli da pokažemo.

U prethodnom izvođenju smo se oslonili na činjenicu da su i a i b brojevi koji su veći ili jednaki od 0 i strogo manji od m. Pronalaženje vrednosti razlike po modulu m moguće je i u opštem slučaju. Naime, za proizvoljne brojeve A i B važi:

$$(B - A) \bmod m = (B \bmod m - A \bmod m + m) \bmod m$$

Dokažimo ovo tvrđenje. Podsetimo se da je $x \bmod m = r$ ako i samo ako postoji q takav da je $x = q \cdot m + r$ i ako je $0 \leq r < m$. Neka je $A = q_a \cdot m + a$ i $B = q_b \cdot m + b$, za $0 \leq a, b < m$. Zato je $A \bmod m = a$ i $B \bmod m = b$. Neka je:

$$b - a + m = p \cdot m + r,\ 0 \leq r < m.$$

Zato je:

$$(B \bmod m - A \bmod m + m) \bmod m = (b - a + m) \bmod m = r.$$

Takođe, važi i da je:

$$B - A = (q_b - q_a) \cdot m + (b - a) = (q_b - q_a - 1) \cdot m + (b - a + m) = (q_b - q_a - 1 + p) \cdot m + r,$$

pa je i

$$(B - A) \bmod m = r,$$

čime je tvrđenje dokazano.

Važi i naredno tvrđenje koje nećemo dokazivati:

$$a^n \bmod m = (a \bmod m)^n \bmod m$$

a koje ima važnu primenu jer za iole veće vrednosti n pri računanju vrednosti a^n može doći do prekoračenja.

Zadatak: Napisati program koji određuje poslednje tri cifre zbira i poslednje tri cifre proizvoda četiri uneta cela broja manja od 1000.

Ideja koja prirodno prva padne na pamet je da se izračunaju zbir i proizvod unetih brojeva, a da se onda poslednje tri cifre odrede izračunavanjem ostatka pri deljenju sa 1000. Kada se u obzir uzme raspon brojeva koji se unose, takvo rešenje bi davalo korektne rezultate u slučaju zbira, međutim, proizvod brojeva može biti mnogo veći od samih brojeva i zato je za izračunavanje poslednje tri cifre proizvoda potrebno primeniti malo napredniji algoritam koji koristi činjenicu da su za poslednje tri cifre proizvoda relevantne samo poslednje tri cifre svakog od činilaca. Zato je pre množenja moguće odrediti ostatak pri deljenju sa 1000 svakog od činilaca, izračunati proizvod ovih ostataka, a zatim odrediti njegove tri poslednje cifre izračunavanjem ostatka pri deljenju sa 1000.

Funkcije za množenje i sabiranje po modulu zasnovaćemo direktno na prikazanim relacijama, a onda ćemo ih primenjivati tako što ćemo proizvod (tj. zbir) po modulu m svih prethodnih brojeva primenom funkcija kombinovati sa novim brojem. Dakle, ako funkciju za sabiranje po modulu m označimo sa $+_m$, a množenje po modulu m sa $\cdot_m$, izračunavaćemo vrednosti izraza $((a +_m b) +_m c) +_m d$ tj. izraza $((a \cdot_m b) \cdot_m c) \cdot_m d$.

Funkcije za sabiranje i množenje po modulu imaju pored brojeva koje treba sabrati, odnosno pomnožiti kao argument i vrednost modula po kome se operacije računaju.

```
int plus_mod(int a, int b, int m){
   return ((a % m) + (b % m)) % m;
}

int puta_mod(int a, int b, int m){
   return ((a % m) * (b % m)) % m;
}

int main(){
   int a, b, c, d;
   cout << "Unesi cetiri broja" << endl;
   cin >> a >> b >> c >> d;
   cout << "Poslednje tri cifre njihovog zbira su "
     << plus_mod(plus_mod(plus_mod(a,b,1000),c,1000),d,1000) << endl;
   cout << "Poslednje tri cifre njihovog proizvoda su "
     << puta_mod(puta_mod(puta_mod(a,b,1000),c,1000),d,1000) << endl;
   return 0;
}
```

Ipak, imajmo u vidu da je izračunavanje celobrojnog količnika i ostatka vremenski zahtevna operacija (najčešće se izvršava dosta sporije nego osnovne aritmetičke operacije), tako da je u praksi poželjno izbeći ih kada je to moguće. Na primer, ako je zbir $a + b$ moguće predstaviti odabranim tipom podataka, tada je umesto

izraza $(a \bmod m + b \bmod m) \bmod m$ ipak bolje koristiti izraz $(a + b) \bmod m$.

Moguća je i naredna optimizacija. Naime, na osnovu invarijante, tekuća vrednost je uvek u rasponu $[0, 1000)$: ovo važi za ulazne vrednosti, a važi i za rezultat funkcija `plus_mod` i `puta_mod`. Stoga nije neophodno računati ostatak tekuće vrednosti pri deljenju sa 1000. Dakle, od tri pojave operacije `%` u funkcijama `plus_mod` i `puta_mod`, dovoljno je ostaviti samo poslednju.

```
int plus_mod(int a, int b, int m){
   return (a + b) % m;
}

int puta_mod(int a, int b, int m){
   return (a * b) % m;
}
```

Rastavljanje broja na proste činioce (faktorizacija)

Problem: Dat je broj n. Rastaviti ga na proste činioce.

Na primer, za $n = 315$, potrebno je odštampati `3 3 5 7`.

Problem možemo rešavati induktivno-rekurzivnim pristupom na sledeći način. Pretpostavimo da sve brojeve manje od n umemo da rastavimo na proste činioce. Ako n nije prost broj, onda se on može predstaviti kao proizvod $n = d \cdot n_1$, gde je d najmanji od svih činilaca broja n (dakle prost broj) i važi $d \leq \sqrt{n}$. Činilac d se onda može pronaći prolaskom kroz skup prirodnih brojeva redom od 2 do $\sqrt{n}$ (redosled nam garantuje pronalaženje najmanjeg činioca), a ostale činioce broja n_1 po induktivnoj hipotezi znamo da odredimo. Na osnovu ove konstrukcije jednostavno je formulisati rekurzivni algoritam.

Iterativna varijanta ovog rekurzivnog algoritma sastojala bi se iz narednih koraka: prolazi se skupom brojeva od $i = 2$ do $\sqrt{n}$ i ako je n deljivo brojem i, sve dok je n deljivo sa i štampamo vrednost i i postavljamo vrednost n na n/i. Specijalan slučaj čini situacija kada je n prost broj, u kom slučaju je potrebno samo odštampati sam taj broj. S obzirom na činjenicu da nijedan paran broj veći od 2 nije paran, efikasnije je zasebno razmatrati broj 2, a zatim proći skupom neparnih brojeva od 3 do $\sqrt{n}$.

```
// funkcija koja ispisuje sve proste cinioce broja n
void ispisiProsteCinioce(int n){

  // ako je n parno, stampamo 2 onoliki broj puta koliko puta deli n
  while (n % 2 == 0){
    cout << 2 << " ";
    // azuriramo n
    n /= 2;
```

```
  }

  // n je neparno
  // prolazimo kroz neparne brojeve
  for (int i = 3; i*i <= n; i = i + 2){
    // sve dok i deli n, stampamo i
    while (n % i == 0){
      cout << i << " ";
      // azuriramo n
      n /= i;
    }
  }

  // ako je n prost broj, samo stampamo n
  if (n > 2)
    cout << n << " ";
  cout << endl;
}

int main(){
  int n;
  cout << "Unesite broj n" << endl;
  cin >> n;
  ispisiProsteCinioce(n);
  return 0;
}
```

Istaknimo jednu važnu stvar: naime, nigde u programu se posebno ne određuju prosti brojevi. Na prvi pogled ovo može delovati zbunjujuće, međutim redosled kojim tražimo činioce nam garantuje da kao činilac nikada nećemo odštampati neki složeni broj $s = i \cdot j$, $i, j < s$ jer ćemo pre ispitivanja deljivosti sa s proveravati deljivost sa i i j, s obzirom na to da je njihova vrednost manja od s.

Primetimo da se za složene brojeve granica `for` petlje smanjuje sa svakim novim pronađenim činiocem (jer će nova vrednost za n biti strogo manja od stare). Ako su prosti činioci (ne nužno različiti) broja n redom jednaki

$$p_1 \leq p_2 \leq \ldots \leq p_{k-1} \leq p_k,$$

tada je složenost ovog algoritma za faktorizaciju $O(\max(p_{k-1}, \sqrt{p_k}))$. Naime, kada pronađemo sve proste činioce osim dva najveća p_{k-1} i p_k, ostajemo sa brojem $n = p_{k-1} \cdot p_k$. Prost činilac p_{k-1} nalazimo posle $O(p_{k-1})$ koraka, nakon čega broj n postaje p_k i petlja po i ide do $\sqrt{p_k}$. Dakle, veća od vrednosti p_{k-1} i $\sqrt{p_k}$ odrediće broj izvršavanja petlje. Kada je broj prost, tada ne postoji p_{k-1} i složenost je jednaka $O(\sqrt{p_k})$. Dakle, u najgorem slučaju složenost prethodnog algoritma za faktorizaciju broja iznosi $O(\sqrt{n})$.

Važi i sledeće: nije jednako teško faktorisati sve brojeve jedne iste dužine. Najteže

instance ovog problema ako se primenjuje opisani algoritam čine brojevi koji su proizvodi dva približno jednaka prosta broja. Kada su oba ova prosta činioca velika i slične veličine, ovaj problem postaje jako teško rešiti. Primenom efikasnijeg algoritma 2009. godine okončao se dvogodišnji napor grupe istraživača da se faktoriše broj od 232 cifre korišćenjem više stotina računara. Pretpostavljena težina ovog problema je osnova za procenu sigurnosti velikog broja kriptografskih algoritama, kao što je RSA algoritam.

Faktorizacija većeg broja brojeva

Ukoliko je potrebno veliki broj puta izvršiti faktorizaciju brojeva, npr. $O(n)$ puta izvršiti faktorizaciju brojeva koji su manji ili jednaki n, onda je potrebno $O(n)$ puta izvršiti ispočetka ovaj algoritam, te bi ukupna složenost prethodnog algoritma iznosila $O(n\sqrt{n})$.

Faktorizaciju brojeva možemo izvršiti i korišćenjem pomoćnog niza dužine n koji za svako $k \leq n$ sadrži najmanji prost činilac broja k. Naime, kad bismo znali vrednost najmanjeg prostog činioca svakog broja k za $k \leq n$, broj m bismo mogli faktorisati na sledeći način: štampamo vrednost p_1 najmanjeg prostog činioca broja m, zatim vrednost p_2 najmanjeg prostog činioca broja m/p_1, zatim vrednost najmanjeg prostog činioca broja $m/(p_1 \cdot p_2), \ldots$ Niz koji sadrži vrednost najmanjeg prostog činioca za svaki broj manji ili jednak n može se dobiti malim prilagođavanjem Eratostenovog sita: umesto da kao u originalnom algoritmu prilikom prolaska kroz umnoške nekog prostog broja označavamo da su ti brojevi složeni, postavljaćemo vrednost najmanjeg prostog činioca tog broja (ukoliko za dati broj već nije postavljen vrednost manjeg činioca). Prilikom razmatranja prostog broja d, možemo da krenemo od d^2 jer su brojevi $2d, 3d, \ldots, (d-1)d$ deljivi redom sa $2, 3, \ldots, d-1$ te sigurno imaju činioca manjeg od d. Nakon toga izračunaćemo proste činioce broja n uzastopnim deljenjem broja n njegovim najmanjim prostim činiocem.

U nastavku je data implementacija algoritma faktorizacije svih brojeva od 1 do n korišćenjem Eratostenovog sita.

```
vector<int> eratosten(int n) {
  // niz koji cemo popuniti tako da se na poziciji i
  // nalazi najmanji prost cinilac broja i
  vector<int> najmanjiCinilac(n + 1);

  // postavljamo da je najmanji prost cinilac svakog broja sam taj broj
  for (int i = 1; i <= n; i++)
    najmanjiCinilac[i] = i;
  for (int d = 2; d * d <= n; d++)
    // ako je broj d prost, tj njegov najmanji prost činilac je on sam
    if (najmanjiCinilac[d] == d)
      // prolazimo kroz skup svih umnozaka tog broja
```

```
      for (int i = d * d; i <= n; i += d)
        // ako nije ranije azurirana vrednost, odnosno
        // postavljena neka manja vrednost za cinilac
        if (najmanjiCinilac[i] == i)
          // postavljamo da je najmanji prost cinilac broj d
          najmanjiCinilac[i] = d;
  return najmanjiCinilac;
}

// funkcija koja ispisuje sve proste cinioce broja n
void ispisiProsteCinioce(int n, const vector<int>& najmanjiCinilac){
  while (n != 1){
    // stampamo najmanji prost cinilac broja n
    cout << najmanjiCinilac[n] << " ";
    // azuriramo vrednost n
    n /= najmanjiCinilac[n];
  }
  cout << endl;
}

int main(){
  int n;
  cout << "Unesite broj n" << endl;
  cin >> n;
  auto najmanjiCinilac = eratosten(n);
  for (int i = 1; i <= n; i++)
    ispisiProsteCinioce(i, najmanjiCinilac);
  return 0;
}
```

Prikažimo kako se ovim algoritmom vrši faktorizacija svih brojeva manjih ili jednakih 50. Krećemo od niza u kome je na poziciji i upisana vrednost i.

```
 1  2  3  4  5  6  7  8  9 10
11 12 13 14 15 16 17 18 19 20
21 22 23 24 25 26 27 28 29 30
31 32 33 34 35 36 37 38 39 40
41 42 43 44 45 46 47 48 49 50
```

U prvom koraku svim parnim brojevima postavljamo da je najmanji prost činilac jednak 2:

```
 1  2  3  2  5  2  7  2  9  2
11  2 13  2 15  2 17  2 19  2
21  2 23  2 25  2 27  2 29  2
31  2 33  2 35  2 37  2 39  2
41  2 43  2 45  2 47  2 49  2
```

U narednom koraku onim umnošcima broja 3 počev od vrednosti 9 koji nisu deljivi sa 2 postavljamo najmanji prost činilac na 3: to su 9, 15, 21, 27, 33, 39 i 45.

```
 1  2  3  2  5  2  7  2  3  2
11  2 13  2  3  2 17  2 19  2
 3  2 23  2 25  2  3  2 29  2
31  2  3  2 35  2 37  2  3  2
41  2 43  2  3  2 47  2 49  2
```

U narednom koraku razmatramo broj 4. Zaključujemo da je složen jer je njegov najmanji prost činilac već postavljen, te njega preskačemo.

U narednom koraku razmatramo umnoške broja 5 počev od vrednosti 25: ako broj nije deljiv nekim manjim prostim brojem (2 i 3), postavljamo da je najmanji prost činilac tog broja jednak 5. To će biti brojevi 25 i 35.

```
 1  2  3  2  5  2  7  2  3  2
11  2 13  2  3  2 17  2 19  2
 3  2 23  2  5  2  3  2 29  2
31  2  3  2  5  2 37  2  3  2
41  2 43  2  3  2 47  2 49  2
```

Broj 6 kao složen preskačemo i za sve brojeve deljive sa 7 počev od 49, a koji nisu deljivi sa 2, 3 niti 5, postavljamo da je 7 njihov najmanji prost činilac (to je samo broj 49).

```
 1  2  3  2  5  2  7  2  3  2
11  2 13  2  3  2 17  2 19  2
 3  2 23  2  5  2  3  2 29  2
31  2  3  2  5  2 37  2  3  2
41  2 43  2  3  2 47  2  7  2
```

Time se obrada završava jer je $8^2 \geq 50$.

Prilikom faktorizacije broja 48 ispisale bi se redom vrednosti $2, 2, 2, 2$ i 3 kao vrednosti najmanjih činilaca brojeva 48, $48/2 = 24$, $24/2 = 12$, $12/2 = 6$ i $6/2 = 3$ redom.

Složenost prvog koraka algoritma – određivanja najmanjeg prostog činioca svih brojeva do n jednaka je složenosti algoritma Eratostenovo sito koja je, ako pretpostavimo da je složenost sabiranja $O(1)$, jednaka $O(n \log \log n)$.[1] Složenost ispisivanja prostih činilaca datog broja iznosi $O(\log n)$ jer je maksimalni broj prostih činilaca broja n jednak $\log_2 n$, a pretpostavljamo da je deljenje činiocima konstantne vremenske složenosti.[2] Dakle, ukoliko je potrebno izvršiti faktorizaciju

[1] Napomenimo da ako bi n moglo biti proizvoljno veliko (ako bi broj bio predstavljen reprezentacijom sa proizvoljnim brojem bitova), onda bi složenost sabiranja bila $O(\log n)$, međutim u konkretnim implementacijama mi smo ograničeni opsegom celobrojnog tipa i sabiranje je složenosti $O(1)$.

[2] Ako bi broj n mogao biti proizvoljno veliki, onda bi složenost deljenja bila $O(\log^2 n)$.

$O(n)$ puta onda će složenost ovog algoritma biti $O(n \log \log n) + O(n \log n) = O(n \log n)$ što je efikasnije od prethodnog pristupa.

Za veliko n prikazani algoritam nije dovoljno efikasan.

Ojlerova funkcija

Ojlerova funkcija (eng. Euler's totient function) ϕ broja n označava broj prirodnih brojeva manjih ili jednakih n koji su uzajamno prosti sa n. Na primer, $\phi(9) = 6$ jer postoji šest brojeva $1, 2, 4, 5, 7, 8$ manjih od 9 koji su uzajamno prosti sa 9, a $\phi(17) = 16$ jer je broj 17 prost i uzajamno je prost sa svim brojevima manjim od njega. Po definiciji važi $\phi(1) = 1$[3].

Ojlerova funkcija ima primenu u teoriji brojeva, u algebri, a takođe igra ključnu ulogu u RSA sistemu za šifrovanje.

Problem: Za dati prirodan broj n izračunati vrednost Ojlerove funkcije $\phi(n)$.

Direktan način da se reši problem bio bi da se redom prođe kroz sve brojeve od 1 do $n - 1$ i da se izbroji koliko je od njih uzajamno prosto sa n.

```
// funkcija koja racuna najveci zajednicki delilac dva broja
int nzd(int a, int b){
  if (a == 0)
    return b;
  return nzd(b % a, a);
}

// funkcija koja racuna vrednost Ojlerove funkcije
int ojlerovaFunkcija(int n){
  // 1 je uvek uzajamno prosto sa n
  int br = 1;
  for (int i = 2; i < n; i++)
    if (nzd(i, n) == 1)
      br++;
  return br;
}

int main(){
  int n;
  cout << "Unesite n" << endl;
  cin >> n;
  cout << "fi(" << n << ")=" << ojlerovaFunkcija(n) << endl;
```

[3]Nekad se Ojlerova funkcija broja n definiše kao broj prirodnih brojeva strogo manjih od n, što jedino utiče na vrednost Ojlerove funkcije broja 1.

```
  return 0;
}
```

Prilikom računanja vrednosti Ojlerove funkcije, funkcija za određivanje najvećeg zajedničkog delioca se poziva $O(n)$ puta. Kako znamo da je složenost algoritma za izračunavanje najvećeg zajedničkog delioca brojeva i i n jednaka $O(\log(i+n))$ i kako je ovde $i < n$, važi da je složenost računanja vrednosti $\text{nzd}(i, n)$ jednaka $O(\log n)$, te ukupna složenost algoritma za izračunavanje vrednosti Ojlerove funkcije broja n iznosi $O(n \log n)$.

Razmotrimo sada efikasnije rešenje ovog problema. Iz same definicije Ojlerove funkcije sledi da je $\phi(1) = 1$ i da je $\phi(p) = p - 1$ za prost broj p. Takođe, ako je $n = p^k$, gde je p neki prost broj onda od p^k brojeva koji manji ili jednaki n sa njim nisu uzajamno prosti samo umnošci broja p, odnosno brojevi $p, 2p, 3p, \ldots, p^{k-1}p$ kojih ima ukupno p^{k-1}. Dakle, važi

$$\phi(p^k) = p^k - p^{k-1} = p^k(1 - \frac{1}{p}) \tag{1}$$

Važi i sledeće: za uzajamno proste brojeve n i m važi[4]:

$$\phi(n \cdot m) = \phi(n) \cdot \phi(m) \tag{2}$$

ako su brojevi n i m uzajamno prosti. Funkcije za koje važi ovo svojstvo nazivamo *multiplikativnim funkcijama* (eng. multiplicative function).[5] Različite funkcije u teoriji brojeva su multiplikativne, što omogućava da se na osnovu vrednosti funkcije za proste brojeve lako izračuna vrednost funkcije za proizvoljnu vrednost argumenta; na primer broj delilaca broja n.

Ako broj n u opštem slučaju predstavimo kao $n = p_1^{k_1} \cdot \ldots \cdot p_r^{k_r}$, gde su $p_1, p_2, \ldots, p_r$ prosti činioci broja n, na osnovu jednačina (1) i (2), sledi:

$$\begin{aligned}
\phi(n) &= \phi(p_1^{k_1})\phi(p_2^{k_2}) \cdot \ldots \cdot \phi(p_r^{k_r}) \\
&= p_1^{k_1}(1 - \frac{1}{p_1})p_2^{k_2}(1 - \frac{1}{p_2}) \cdot \ldots \cdot p_r^{k_r}(1 - \frac{1}{p_r}) \\
&= p_1^{k_1}p_2^{k_2} \cdot \ldots \cdot p_r^{k_r}(1 - \frac{1}{p_1})(1 - \frac{1}{p_2}) \cdot \ldots \cdot (1 - \frac{1}{p_r}) \\
&= n \cdot \prod_{\substack{p|n \\ p \text{ prost}}} (1 - \frac{1}{p})
\end{aligned} \tag{3}$$

[4]Dokaz ovog tvrđenja sledi iz kineske teoreme o ostacima koju ćemo kasnije razmatrati.

[5]Preciznije, da bi funkcija f bila multiplikativna potrebno je i da važi $f(1) = 1$ što jeste tačno u ovom slučaju kad smo Ojlerovu funkciju broja n definisali kao broj prirodnih brojeva manjih ili jednakih od n. Čak i ako bi se razmatrala alternativna definicija Ojlerove funkcije po kojoj je $\phi(1) = 0$, dalje izvođenje bilo bi tačno.

Na primer $\phi(36) = \phi(2^2 \cdot 3^2) = 36(1 - \frac{1}{2})(1 - \frac{1}{3}) = 36 \cdot \frac{1}{2} \cdot \frac{2}{3} = 12$

```
// funkcija koja racuna vrednost Ojlerove funkcije
int ojlerovaFunkcija(int n){
  // rezultat inicijalizujemo na n
  int fi = n;
  // za svaki prost cinilac p broja n
  // rezultat mnozimo sa 1-1/p = (p-1)/p
  // usput azuriramo vrednost broja n
  for (int p = 2; p * p <= n; p++){
    if (n % p == 0) {
      while (n % p == 0)
        n /= p;
      fi = fi * (p-1)/p;
    }
  }
  // ako je n prost broj
  if (n > 1)
    fi = fi * (n-1)/n;
  return fi;
}
```

Ovim smo problem računanja Ojlerove funkcije broja n sveli na problem faktorizacije broja n. Složenost ovog algoritma odgovara složenosti odgovarajućeg algoritma za faktorizaciju broja, odnosno iznosi $O(\sqrt{n})$.

S obzirom na to da važi

$$fi \cdot \frac{p-1}{p} = fi \cdot (1 - \frac{1}{p}) = fi - \frac{fi}{p},$$

možemo izbeći množenje u prethodnoj implementaciji.

```
int ojlerovaFunkcija(int n){
  // rezultat inicijalizujemo na n
  int fi = n;
  // za svaki prost cinilac p oduzimamo od fi broj umnozaka cinioca p
  for (int p = 2; p * p <= n; p++){
    if (n % p == 0){
      while (n % p == 0)
        n /= p;
      fi -= fi / p;
    }
  }
  // ako je n prost
  if (n > 1)
    fi -= fi / n;
  return fi;
}
```

Umesto da prolazi skupom svih prirodnih brojeva od 2 do $\sqrt{n}$, vrednost p može uzeti vrednost 2, a zatim prolaziti skupom neparnih brojevima, čime bi se dobilo ubrzanje za konstantan faktor.

Računanje Ojlerove funkcije većeg broja brojeva

Ukoliko bi zadatak bio da se izračuna vrednost Ojlerove funkcije za sve brojeve manje ili jednake n, prethodna implementacija bila bi složenosti $O(n\sqrt{n})$. Kao i ranije, razmotrimo varijantu algoritma zasnovanu na Eratostenovom situ kojom se angažuje dodatni memorijski prostor veličine $O(n)$. Na osnovu jednačine (3) možemo zaključiti da se za sve brojeve deljive nekim prostim brojem p u izrazu za Ojlerovu funkciju javlja činilac $1 - \frac{1}{p}$. Dakle, možemo redom proći kroz sve proste brojeve i vrednosti Ojlerove funkcije svih umnožaka tekućeg prostog broja p pomnožiti izrazom $1 - \frac{1}{p}$.

Inicijalizujmo sve vrednosti pomoćnog niza $\phi(i)$, $1 \leq i \leq n$, na i. Razmatraćemo redom vrednosti za p počev od 2 do n: naime, za razliku od osnovne varijante Eratostenovog sita gde je vrednost za p išla do $\sqrt{n}$ ovde je neophodno da p prođe skupom svih vrednosti do n zbog postavljanja vrednosti Ojlerove funkcije svih prostih brojeva p na $p - 1$. Ukoliko je $\phi(p) = p$, to znači da je broj p prost i vrednost Ojlerove funkcije svih umnožaka broja p pomnožićemo sa $\frac{p-1}{p}$. Pritom ne možemo kao u originalnoj verziji Eratostenovog sita krenuti od vrednosti p^2 jer moramo sve umnoške broja p (i $2p, 3p, \ldots, (p-1)p$) pomnožiti sa $\frac{p-1}{p}$. Na kraju algoritma u pomoćnom nizu $\phi(i)$ naći će se vrednosti Ojlerove funkcije za sve vrednosti i od 1 do n.

```
void ispisiVrednostiOjleroveFunkcijeDoN(int n){
  // niz u kome na poziciji i formiramo vrednost fi(i)
  vector<int> fi(n + 1);
  // inicijalizujemo sve vrednosti
  for (int i = 1; i <= n; i++)
    fi[i] = i;

  // za sve vrednosti od 2 do n
  for (int p = 2; p <= n; p++){
    // ako vrednost nije menjana, to znaci da je p prosto
    if (fi[p] == p){
      // postavljamo vrednost Ojlerove funkcije za prost broj p
      fi[p] = p - 1;

      // azuriramo vrednosti Ojlerove funkcije
      // za sve umnoske broja p
      for (int i = 2 * p; i <= n; i += p){
        fi[i] = (fi[i]/p) * (p - 1);
      }
```

```
    }
  }
  // stampamo dobijene vrednosti Ojlerove funkcije
  for (int i = 1; i <= n; i++)
    cout << "fi(" << i << ")=" << fi[i] << endl;
}

int main(){
    int n;
    cout << "Unesite n" << endl;
    cin >> n;
    ispisiVrednostiOjleroveFunkcijeDoN(n);
    return 0;
}
```

Prikažimo kako se ovim algoritmom određuje vrednost Ojlerove funkcije svih brojeva manjih ili jednakih 20. Krećemo od niza u kome je na poziciji i upisana vrednost i.

```
 1  2  3  4  5  6  7  8  9 10
11 12 13 14 15 16 17 18 19 20
```

U prvom koraku se vrednost Ojlerove funkcije broja 2 postavlja na $2 - 1 = 1$, a za sve brojeve deljive sa 2 vrednost Ojlerove funkcije se množi sa $1 - 1/2 = 1/2$.

```
 1  1  3  2  5  3  7  4  9  5
11  6 13  7 15  8 17  9 19 10
```

Nakon toga se razmatra prost broj 3, postavlja se $\phi(3) = 2$, a za sve brojeve deljive sa 3 vrednost Ojlerove funkcije množi se sa $1 - 1/3 = 2/3$.

```
 1  1  2  2  5  2  7  4  6  5
11  4 13  7 10  8 17  6 19 10
```

Broj 4 se preskače kao složen, zatim se postavlja $\phi(5) = 4$ i za sve brojeve deljive sa 5 vrednost Ojlerove funkcije se množi sa $1 - 1/5 = 4/5$.

```
 1  1  2  2  4  2  7  4  6  4
11  4 13  7  8  8 17  6 19  8
```

Broj 6 se preskače kao složen, a zatim se razmatra prost broj 7. Postavlja se $\phi(7) = 6$ i za sve brojeve deljive sa 7 vrednost Ojlerove funkcije se množi sa $1 - 1/7 = 6/7$.

```
 1  1  2  2  4  2  6  4  6  4
11  4 13  6  8  8 17  6 19  8
```

Brojevi 8, 9 i 10 se preskaču kao složeni, a brojevima 11, 13, 17 i 19 se (kao prostim brojevima) postavlja vrednost Ojlerove funkcije. Ne postoje brojevi manji od 20 koji su njima deljivi te nemamo dodatnih ažuriranja.

```
 1  1  2  2  4  2  6  4  6  4
```

```
10  4 12  6  8  8 16  6 18  8
```

Napomenimo da iako se sa prolazom kroz činioce ne ide do $\sqrt{n}$ već do n, složenost algoritma odgovara složenosti Eratostenovog sita, odnosno iznosi $O(n \log \log n)$.

Jedno važno svojstvo Ojlerove funkcije iskazuje Ojlerova teorema.

Ojlerova teorema: Ako je $n \geq 1$ i a je ceo broj uzajamno prost sa n, onda važi

$$a^{\phi(n)} \equiv 1 \pmod{n}.$$

Ako u tvrđenje Ojlerove teoreme umesto n uvrstimo prost broj p i ako a nije deljivo sa p, s obzirom da važi $\phi(p) = p - 1$ važiće $a^{p-1} \equiv 1 \pmod{p}$. Množenjem obe strane jednakosti sa a dobija se tvrđenje Male Fermaove teoreme.

Mala Fermaova teorema: Ako je p prost broj, onda za svaki ceo broj a važi

$$a^p \equiv a \pmod{p}$$

Dakle, Olerova teorema predstavlja uopštenje Male Fermaove teoreme. Naravno, tvrđenje male Fermaove teoreme trivijalno važi i u slučaju kada je a deljivo sa p.

Jedna primena Male Fermaove teoreme je za testiranje da li je dati broj n prost ili ne (eng. Fermat primality test). Naime, za dati broj n možemo izabrati proizvoljan broj a koji je uzajamno prost sa n i izračunati vrednost $a^{n-1} \bmod n$. Ukoliko je rezultat različit od 1, broj n je sigurno složen. Ukoliko je jednak 1, pokazuje se da je mala verovatnoća da broj n nije prost. Naime, ako se za jednu ili veći broj vrednosti za a pokaže da je vrednost izraza $a^{n-1} \bmod n$ jednaka 1, broj n je verovatno prost.

Pretpostavimo da je n jednako proizvodu dva različita prosta broja p i q. Problem faktorizacije broja $n = p \cdot q$ je težak, međutim ako je pored n poznata i vrednost $\phi(n)$, onda se brojevi p i q mogu jednostavno odrediti. Naime pošto su brojevi p i q prosti, oni su i uzajamno prosti pa važi $\phi(n) = (p-1) \cdot (q-1)$. Odavde dobijamo da je $n - \phi(n) + 1 = p + q$. Na osnovu vrednosti $p + q$ i $p \cdot q$ lako se mogu odrediti brojevi p i q kao rešenja kvadratne jednačine

$$x^2 - (n - \phi(n) + 1)x + n = 0$$

Ovo je važna stvar u razmatranju sigurnosti RSA enkripcije. Naime, jačina metoda enkripcije se zasniva na tajnosti vrednosti Ojlerove funkcije $\phi(n)$, a kao što smo prethodno videli računanje vrednosti Ojlerove funkcije odgovara faktorizaciji broja, što se smatra teškim problemom.

Funkcije delilaca

Funkcija delilaca (eng. divisor function) je neka funkcija nad skupom delilaca datog celog broja. U funkcije delilaca spadaju broj delilaca i suma delilaca broja koje ćemo detaljnije razmatrati.

Broj delilaca

Problem: Za dati broj n izračunati ukupan broj njegovih delilaca.

Na primer, ako je $n = 24$, njegovi delioci su $1, 2, 3, 4, 6, 8, 12$ i 24 te je tražena vrednost 8.

Primetimo da broj n uvek deli sam sebe te je broj delilaca broja n uvek veći ili jednak 1. Naivni pristup za određivanje broja delilaca broja n bi prolazio skupom svih brojeva od 1 do $n/2$ i za svaku vrednost koja deli broj n ukupan broj delilaca bi se uvećao za 1.

```
// funkcija koja racuna broj delilaca broja n
// razmatrajuci sve moguce delioce pojedinacno
int brojDelilaca(int n){
  // broj delilaca je bar 1 -- sam taj broj
  int br = 1;
  for (int i = 1; i <= n/2; i++)
    if (n % i == 0)
      br++;
  return br;
}
```

Složenost ovog algoritma iznosi $O(n)$.

Primetimo da se delioci uglavnom javljaju u paru: u prethodnom primeru imamo da je $24 = 1 \cdot 24 = 2 \cdot 12 = 3 \cdot 8 = 4 \cdot 6$. Ovo ne važi samo kada je n kvadrat nekog broja: npr. delioci broja 16 su $1, 2, 4, 8$ i 16 gde središnji delilac 4 nema svog para. Nešto efikasniji algoritam bi prolazio skupom svih brojeva manjih ili jednakih $\sqrt{n}$ i za svaki broj d koji deli broj n ako je $d \neq \frac{n}{d}$ ukupan broj delilaca bi se uvećao za 2, a ako je $d = \frac{n}{d}$ za 1.

```
// funkcija koja racuna broj delilaca broja n
// razmatrajuci delioce u paru
int brojDelilaca(int n){
  int br = 0;
  for (int i = 1; i * i <= n; i++)
    if (n % i == 0)
      if (i * i != n)
        br += 2;
      else
        br++;
  return br;
}
```

Složenost prethodnog algoritma iznosi $O(\sqrt{n})$.

Neka je $n = p_1^{k_1} \cdot \ldots \cdot p_r^{k_r}$, gde su $p_1, p_2, \ldots, p_r$ svi različiti prosti činioci broja n. Prosti činioci svakog delioca broja n moraju biti iz skupa $p_1, p_2, \ldots, p_r$, te je opšti

oblik svakog delioca broja n jednak $d = p_1^{l_1} \cdot \ldots \cdot p_r^{l_r}$, gde za svako $1 \leq i \leq r$ važi $0 \leq l_i \leq k_i$. Dakle, za eksponent činioca p_i postoji $k_i + 1$ različitih mogućnosti, te je ukupan broj delilaca broja n jednak $(k_1 + 1)(k_2 + 1) \cdot \ldots \cdot (k_r + 1)$. Dakle, ako znamo faktorizaciju broja n, broj delilaca možemo lako efektivno odrediti.

Razmotrimo slučaj broja $24 = 2^3 \cdot 3^1$. Svi njegovi delioci su $2^0 \cdot 3^0$, $2^0 \cdot 3^1$, $2^1 \cdot 3^0$, $2^1 \cdot 3^1$, $2^2 \cdot 3^0$, $2^2 \cdot 3^1$, $2^3 \cdot 3^0$ i $2^3 \cdot 3^1$ ima ih $(3 + 1) \cdot (1 + 1) = 8$.

```
int brojDelilaca(int n) {
  // ukupan broj delilaca broja n
  int broj = 1;
  // prolazimo skupom svih brojeva od 2 do sqrt(n)
  for (int i = 2; i * i <= n; i++) {
    // broj puta koliko i deli n
    int k = 0;
    while (n % i == 0) {
      n /= i;
      k++;
    }
    // ukupan broj mnozimo sa k+1
    broj *= (k + 1);
  }
  // ako je preostalo n prost broj, njegova vrednost za k je 1,
  // te je prethodni rezultat potrebno pomnoziti sa k + 1 = 2
  if (n > 1)
    broj *= 2;

  return broj;
}
```

Složenost prethodnog algoritma odgovara složenosti faktorizacije broja n, te iznosi $O(\sqrt{n})$.

Suma delilaca

Problem: Za dati broj n izračunati sumu svih njegovih delilaca.

Na primer, ako je $n = 24$, njegovi delioci su $1, 2, 3, 4, 6, 8, 12$ i 24 i njihov zbir jednak je 60.

Naivni pristup za rešavanje ovog problema bi prolazio skupom svih brojeva manjih od n i uvećavao tekuću sumu delilaca za vrednost svakog od brojeva koji deli broj n. Složenost ovog pristupa iznosila bi $O(n)$.

```
// funkcija koja racuna sumu delilaca broja n
// razmatrajuci sve moguce delioce pojedinacno
int sumaDelilaca(int n){
  // n je uvek sam sebi delilac
```

```
  int suma = n;
  for (int i = 1; i < n; i++)
    if (n % i == 0)
      suma += i;
  return suma;
}
```

Efikasniji pristup bi delioce otkrivao u paru, tako što bi prolazio skupom svih brojeva manjih ili jednakih $\sqrt{n}$ i za svaki broj i koji deli broj n vrednost i i vrednost njemu odgovarajućeg činioca n/i bi bila dodata na tekuću sumu, osim kada je baš $i = n/i$, odnosno kada je $i = \sqrt{n}$ kada bi se na sumu dodala samo vrednost i.

```
// funkcija koja racuna sumu delilaca broja n
// razmatrajuci delioce u paru
int sumaDelilaca(int n){
  int suma = 0;
  for (int i = 1; i * i <= n; i++)
    if (n % i == 0)
      if (i * i != n)
        suma += (i + n/i);
      else
        suma += i;
  return suma;
}
```

Složenost prethodnog algoritma iznosila bi $O(\sqrt{n})$.

Da li bismo mogli, kao u slučaju računanja broja delilaca, da ako znamo faktorizaciju broja n na neki način efektivno izračunamo sumu delilaca broja n? Neka je $n = p_1^{k_1} \cdot \ldots \cdot p_r^{k_r}$, gde su $p_1, p_2, \ldots, p_r$ svi različiti prosti činioci broja n. Opšti oblik delioca broja n jednak je $d = p_1^{l_1} \cdot \ldots \cdot p_r^{l_r}$, gde važi $0 \leq l_i \leq k_i$, te postoji $k_i + 1$ mogućnosti za eksponent činioca p_i. Primećujemo da se posle oslobađanja od zagrada svaki delilac broja n pojavljuje tačno jednom u narednom izrazu:

$$S = (1 + p_1 + p_1^2 + \ldots + p_1^{k_1}) \cdot (1 + p_2 + p_2^2 + \ldots + p_2^{k_2}) \cdot \ldots \cdot (1 + p_r + p_r^2 + \ldots + p_r^{k_r})$$

Gornji izraz može se pojednostaviti računanjem zbirova odgovarajućih geometrijskih progresija:

$$S = \frac{p_1^{k_1+1} - 1}{p_1 - 1} \cdot \frac{p_2^{k_2+1} - 1}{p_2 - 1} \cdot \ldots \cdot \frac{p_r^{k_r+1} - 1}{p_r - 1}$$

Dakle, problem se svodi na identifikovanje prostih činalaca broja n, odnosno na faktorizaciju broja n.

```
int sumaDelilaca(int n) {
  // ukupna suma delilaca broja n
  int suma = 1;
```

```
  // prolazimo skupom svih brojeva od 2 do sqrt(n)
  for (int i = 2; i * i <= n; i++) {
    // broj puta koliko i deli n
    int k = 0;
    while (n % i == 0) {
      n /= i;
      k++;
    }
    // azuriramo vrednost sume po prethodnoj formuli
    suma *= (pow(i,k+1) - 1) / (i - 1);
  }
  // ako je preostalo n prost broj, za njega vazi k = 1
  // pa njegov doprinos sumi iznosi (n^2-1)/(n-1)
  if (n > 1)
    suma *= (n*n - 1) / (n - 1);
  return suma;
}
```

Složenost algoritma zasnovanog na faktorizaciji je $O(\sqrt{n})$.

Efikasnija rešenja problema izračunavanja broja i sume delilaca zasnivaju se na činjenici da su odgovarajuće funkcije, broj delilaca i suma delilaca, multiplikativne. Dakle, ako sa $B(n)$ označimo broj delilaca broja n, a sa $S(n)$ sumu njegovih delilaca, onda za uzajamno proste brojeve x i y važi $B(x \cdot y) = B(x) \cdot B(y)$ i $S(x \cdot y) = S(x) \cdot S(y)$ jer se svaki činilac broja x može iskombinovati sa svakim činiocem broja y i na taj način dobijamo sve moguće činioce broja xy. U opštem slučaju funkcija delilaca $\sigma_k(n)$ označava sumu k-tih stepena svih delilaca broja n i za nju važi da je multiplikativna. Specijalno, $\sigma_0(n)$ odgovara broju delilaca broja n, a $\sigma_1(n)$ sumi delilaca broja n.

Prošireni Euklidov algoritam

Podsetimo se ideje osnovnog Euklidovog algoritma za određivanje najvećeg zajedničkog delioca brojeva a i b: polazeći od brojeva $r_0 = a$ i $r_1 = b$, izračunava se ostatak $r_2 = r_0 \bmod r_1$, zatim ostatak $r_3 = r_1 \bmod r_2$, itd. Na taj način se dobija opadajući niz ostataka

$$r_{i-1} = q_i r_i + r_{i+1}, \quad 0 \leq r_{i+1} < r_i, \text{ za } i = 1, 2, \ldots, k \tag{4}$$

(pri deljenju r_{i-1} sa r_i količnik je q_i, a ostatak r_{i+1}). Dobijeni niz r_i je konačan jer je opadajući (važi $r_{i+1} < r_i$ za svako i), a sastoji se od prirodnih brojeva. Neka je npr. $r_{k+1} = 0$ i neka je $r_k \neq 0$ poslednji član ovog niza različit od nule. Kako je

$$\text{nzd}(r_0, r_1) = \text{nzd}(r_1, r_2) = \ldots = \text{nzd}(r_{k-1}, r_k) = \text{nzd}(r_k, 0) = r_k,$$

vidimo da je $d = \text{nzd}(a, b)$ upravo jednako r_k, poslednjem ostatku različitom od nule.

Uz malu dopunu, Euklidov algoritam se može iskoristiti i za rešavanje sledećeg problema.

Problem: Najveći zajednički delilac d dva prirodna broja a i b izraziti kao njihovu celobrojnu linearnu kombinaciju. Drugim rečima, odrediti cele brojeve x i y, tako da važi $d = \text{nzd}(a, b) = x \cdot a + y \cdot b$.

Cilj je broj d izraziti u obliku celobrojne linearne kombinacije ostataka $r_0 = a$ i $r_1 = b$, odnosno kao

$$d = r_k = x \cdot r_0 + y \cdot r_1$$

Problem se može rešiti indukcijom. Polazi se od baze, izraza za d u obliku linearne kombinacije dva uzastopna ostatka r_{k-2} i r_{k-1}:

$$d = r_k = r_{k-2} - q_{k-1} r_{k-1}$$

Ovaj izraz je ekvivalentan pretposlednjem deljenju u Euklidovom algoritmu (izraz (4) za $i = k - 1$).

Korak indukcije bio bi da se polazeći od izraza

$$d = x' \cdot r_i + y' \cdot r_{i+1} \tag{5}$$

kao celobrojne linearne kombinacije ostataka r_i i r_{i+1}, broj d izrazi u obliku celobrojne linearne kombinacije ostataka r_{i-1} i r_i, odnosno kao

$$d = x'' \cdot r_{i-1} + y'' \cdot r_i.$$

Zaista, zamenom vrednosti r_{i+1} u jednačini (5) sa $r_{i-1} - q_i r_i$, dobija se

$$d = x' \cdot r_i + y' \cdot r_{i+1} = x' \cdot r_i + y' \cdot (r_{i-1} - q_i r_i) = y' \cdot r_{i-1} + (x' - q_i y') \cdot r_i \tag{6}$$

odnosno:

$$\begin{aligned} x'' &= y' \\ y'' &= x' - q_i y' \end{aligned} \tag{7}$$

tj. dobija se izraz za d u obliku celobrojne linearne kombinacije ostataka r_{i-1} i r_i. Dakle, indukcijom po i, $i = k-2, k-3, \ldots, 1$, dokazano je da se d može izraziti kao celobrojna linearna kombinacija bilo koja dva uzastopna člana r_i i r_{i+1} niza ostataka. Specijalno, za $i = 0$, dobija se traženi izraz. Ova verzija Euklidovog algoritma naziva se *prošireni Euklidov algoritam* (eng. extended Euclidean algorithm).

Primer: Odrediti cele brojeve x i y tako da važi $3 = 33x + 24y$. S obzirom na to da je $\text{nzd}(33, 24) = 3$, možemo iskoristiti prethodni postupak za određivanje brojeva x i y. Prilikom izračunavanja najvećeg zajedničkog delioca brojeva 33 i 24 imamo naredni niz jednakosti: $\text{nzd}(33, 24) = \text{nzd}(24, 9) = \text{nzd}(9, 6) = \text{nzd}(6, 3) = \text{nzd}(3, 0) = 3$. Prateći ovaj niz jednakosti dobijamo: $d = 3 = 9 - 6 = 9 - (24 - 2 \cdot 9) = 3 \cdot 9 - 24 = 3 \cdot (33 - 24) - 24 = 3 \cdot 33 - 4 \cdot 24$, odnosno $x = 3, y = -4$.

Prethodna induktivna konstrukcija pogodna je za rekurzivnu implementaciju jer nove vrednosti za promenljive x i y dobijamo na osnovu prethodnih. Rekurzivna verzija proširenog Euklidovog algoritma imala bi narednu formu.

```
// funkcija koja vraca nzd brojeva a i b i racuna koeficijente x i y
int nzdProsireni(int a, int b, int &x, int &y){
  // ako je b jednako 0, onda je nzd(a,b) = a
  if (b == 0){
    x = 1;
    y = 0;
    return a;
  }

  int x1, y1;
  // rekurzivno resavamo problem za naredna dva elementa u nizu ostataka,
  // a to su brojevi b i a%b
  int nzd = nzdProsireni(b, a % b, x1, y1);
  // na osnovu jednacine (7) vazi naredna veza
  x = y1;
  y = x1 - (a / b) * y1;
  return nzd;
}

int main(){
  int a, b;
  int x, y;
  cout << "Unesite brojeve a i b" << endl;
  cin >> a >> b;
  int d = nzdProsireni(a, b, x, y);
  cout << x << "*" << a << " + " << y << "*" << b
       << " = " << d << endl;
  return 0;
}
```

U prethodnom izvođenju smo uvek $d = r_k$ predstavljali kao linearnu kombinaciju dva susedna elementa iz niza ostataka počev od r_{k-1} i r_{k-2}. Alternativno, moguće je jednu po jednu vrednost iz niza ostataka počev od r_0 predstaviti kao linearnu kombinaciju brojeva a i b i na kraju dobiti r_k. Naime, važi:

$$\begin{aligned} r_0 &= a = 1 \cdot a + 0 \cdot b \\ r_1 &= b = 0 \cdot a + 1 \cdot b \end{aligned}$$

odnosno za $r_0 = a$ važi da je $x = 1$ i $y = 0$, a za $r_1 = b$ važi $x = 0$ i $y = 1$. Želimo da r_{i+1} predstavimo kao celobrojnu linearnu kombinaciju brojeva a i b, ako znamo da važi

$$\begin{aligned} r_{i-1} &= x_{i-1} \cdot a + y_{i-1} \cdot b \\ r_i &= x_i \cdot a + y_i \cdot b \end{aligned}$$

Na osnovu veze $r_{i+1} = r_{i-1} - q_i \cdot r_i$ dobijamo:

$$r_{i+1} = (x_{i-1} \cdot a + y_{i-1} \cdot b) - q_i(x_i \cdot a + y_i \cdot b) = (x_{i-1} - q_i x_i) \cdot a + (y_{i-1} - q_i y_i) \cdot b$$

Dakle, vrednosti x_{i+1} i y_{i+1} možemo izračunati na osnovu prethodnih vrednosti x_{i-1} i x_i, odnosno y_{i-1} i y_i, tj. važi:

$$\begin{aligned} x_{i+1} &= x_{i-1} - q_i x_i \\ y_{i+1} &= y_{i-1} - q_i y_i \end{aligned}$$

Na osnovu ovih veza možemo doći do narednog iterativnog algoritma.

```
// funkcija koja vraca nzd brojeva a i b i racuna koeficijente x i y
int nzdProsireni(int a, int b, int &x, int &y){

  // vrednost x i y za r_0 = a
  int x_preth = 1;
  int y_preth = 0;
  // vrednost x i y za r_1 = b
  int x_tek = 0;
  int y_tek = 1;

  while (b != 0){
    // azuriramo vrednosti za a i b
    // kao u standardnom Euklidovom algoritmu
    int q = a/b;
    int r = a - q * b;
    a = b;
    b = r;
```

```
    // azuriramo tekucu i prethodnu vrednost niza x
    int xpom = x_preth - q * x_tek;
    x_preth = x_tek;
    x_tek = xpom;

    // azuriramo tekucu i prethodnu vrednost niza y
    int ypom = y_preth - q * y_tek;
    y_preth = y_tek;
    y_tek = ypom;
  }

  // postavljamo konacne vrednosti za x i y
  // kao vrednosti x i y za r_k
  x = x_preth;
  y = y_preth;

  // vracamo a kao nzd brojeva
  return a;
}

int main(){
  int a, b;
  int x, y;
  cout << "Unesite brojeve a i b" << endl;
  cin >> a >> b;
  int d = nzdProsireni(a, b, x, y);
  cout << x << "*" << a << " + " << y << "*" << b
       << " = " << d << endl;
  return 0;
}
```

Na primer:

$$
\begin{aligned}
r_0 &= 33 = 1 \cdot 33 + 0 \cdot 24 \\
r_1 &= 24 = 0 \cdot 33 + 1 \cdot 24 \\
r_2 &= 9 = 33 - 24 = (1 \cdot 33 + 0 \cdot 24) - (0 \cdot 33 + 1 \cdot 24) = 1 \cdot 33 - 1 \cdot 24 \\
r_3 &= 6 = 24 - 2 \cdot 9 = (0 \cdot 33 + 1 \cdot 24) - 2 \cdot (1 \cdot 33 - 1 \cdot 24) = -2 \cdot 33 + 3 \cdot 24 \\
r_4 &= 3 = 9 - 6 = (1 \cdot 33 - 1 \cdot 24) - (-2 \cdot 33 + 3 \cdot 24) = 3 \cdot 33 - 4 \cdot 24
\end{aligned}
$$

Broj operacija koje se izvršavaju u proširenom Euklidovom algoritmu proporcionalan je broju operacija u standardnom Euklidovom algoritmu, tj. iznosi $O(\log(a+b))$.

Jedna od primena proširenog Euklidovog algoritma je u rešavanju jedne klase Diofantovih jednačina, tzv. *linearnih Diofantovih jednačina* koje su oblika

$a \cdot x + b \cdot y = c$, gde su a, b, c dati celi brojevi, a x i y su celi brojevi koje treba odrediti. Bez smanjenja opštosti može se pretpostaviti da je $a, b > 0$. Da bi navedena jednačina imala bar jedno rešenje, potrebno je da $d = \text{nzd}(a, b)$ deli c – naime, pošto d deli levu stranu jednačine, mora da deli i desnu. Ako je ovaj uslov ispunjen, jedno od rešenja lako se dobija narednim postupkom: pošto se d izrazi u obliku

$$d = x' \cdot a + y' \cdot b$$

množenjem leve i desne strane jednakosti celim brojem c/d dobija se:

$$c = \frac{x'c}{d} \cdot a + \frac{y'c}{d} \cdot b$$

tj. jedno rešenje jednačine predstavlja par

$$(x, y) = (\frac{x'c}{d}, \frac{y'c}{d})$$

Kao primer, možemo razmotriti rešavanje jednačine $3x + 10y = 7$ i jednačine $4x + 10y = 7$.