Modularni multiplikativni inverz

Multiplikativni inverz broja a po modulu m (eng. modular multiplicative inverse) je broj x za koji važi $a \cdot x \equiv 1 \pmod{m}$. Na primer, multiplikativni inverz broja 5 po modulu 8 je 5 jer važi $5 \cdot 5 \equiv 1 \pmod{8}$. Najčešće se kao multiplikativni inverz broja po modulu m razmatra vrednost iz skupa $\{0, 1, 2, \ldots, m-1\}$.

Postavlja se pitanje da li modularni multiplikativni inverz broja uvek postoji. Razmotrimo slučaj brojeva $m = 8$ i $a = 2$: proverom svih vrednosti po modulu m pokazuje se da modularni multiplikativni inverz broja 2 po modulu 8 ne postoji. Važi naredno tvrđenje:

Teorema: Ako su brojevi a i m uzajamno prosti pozitivni celi brojevi onda multiplikativni inverz broja a po modulu m postoji i on je jedinstven po modulu m. Ako brojevi a i m nisu uzajamno prosti, onda multiplikativni modularni inverz ne postoji.

Dokaz: Pretpostavimo najpre da su brojevi a i m uzajamno prosti. Razmotrimo niz od m brojeva $0, a, 2a, \ldots, (m-1)a$. Pokazaćemo da su sve ove vrednosti različite po modulu m. S obzirom na to da ukupno postoji tačno m različitih vrednosti po modulu m, odatle će slediti da je $x \cdot a \equiv 1 \pmod{m}$ za tačno jedno x iz skupa $\{0, 1, \ldots, m-1\}$ i to x je jedinstveni modularni multiplikativni inverz broja a po modulu m.

Da bismo pokazali ovo tvrđenje, pretpostavimo suprotno, da je $k \cdot a \equiv l \cdot a \pmod{m}$ za dve različite vrednosti k i l iz opsega $0 \leq k, l \leq m-1$. Tada bi važilo $(k-l)a \equiv 0 \pmod{m}$, odnosno $(k-l)a = sm$ za neki ceo broj s. Međutim, s obzirom na to da su a i m uzajamno prosti brojevi, sledi da broj $k-l$ mora biti celobrojni umnožak broja m. Ovo, ipak, nije moguće jer su k i l dva različita cela broja manja od m. Dakle, sve vrednosti $0, a, \ldots, (m-1)a$ su međusobno različite po modulu m i za tačno jednu vrednost iz skupa $\{0, 1, \ldots, m-1\}$ važi da je multiplikativni inverz broja a po modulu m.

Ako brojevi a i m nisu uzajamno prosti, odnosno ako važi $\text{nzd}(a, m) = d > 1$, onda je broj $a \cdot x$ za svako x deljiv sa $d > 1$, pa ni za jedno x ne važi kongruencija $a \cdot x \equiv 1 \pmod{m}$, tj. broj a nema multiplikativni inverz po modulu m. □

Problem: Za data dva uzajamno prosta pozitivna cela broja a i m odrediti multiplikativni inverz broja a po modulu m.

Algoritam grube sile

Naivni pristup rešavanju ovog problema bi za x razmatrao redom sve brojeve od 1 do m i proveravao da li je ostatak pri deljenju broja $a \cdot x$ sa m jednak 1.

```
// direktno racunanje multiplikativnog inverza broja a po modulu m
int modInverz(int a, int m){
```

```
  a = a % m;
  // proveravamo redom kandidate od 1 do m-1
  for (int x = 1; x < m; x++)
    if ((a*x) % m == 1)
      return x;
}

int main(){
  int a, m;
  cout << "Unesite broj a i broj m" << endl;
  cin >> a >> m;
  cout << "Modularni multiplikativni inverz broja " << a
       << " po modulu " << m << " je " << modInverz(a, m) << endl;
  return 0;
}
```

Složenost ovog algoritma je $O(m)$.

Algoritam zasnovan na proširenom Euklidovom algoritmu

Za konstrukciju efikasnijeg algoritma možemo iskoristiti prošireni Euklidov algoritam. Naime, prošireni Euklidov algoritam za dva data broja a i b određuje njihov najveći zajednički delilac d i cele brojeve x i y takve da važi:

$$a \cdot x + b \cdot y = d \tag{1}$$

Da bismo pronašli multiplikativni inverz broja a po modulu m, u jednačinu (1) umesto b uvrstićemo m. Pretpostavka je da su brojevi a i m uzajamno prosti te je $d = 1$. Dakle važi jednakost:

$$a \cdot x + m \cdot y = 1,$$

odakle sledi odgovarajuća kongruencija:

$$x \cdot a + y \cdot m \equiv 1 \pmod{m}.$$

Drugi sabirak na levoj strani jednačine je deljiv sa m pa ga možemo ukloniti, te dobijamo:

$$x \cdot a \equiv 1 \pmod{m}$$

odakle sledi da je multiplikativni inverz broja a po modulu m jednak x. S obzirom na to da dobijena vrednost za x može biti i negativna i veća od m, ukoliko želimo da x zadovoljava uslov $0 \leq x < m$, onda umesto x kao modularni multiplikativni inverz vraćamo vrednost $(x \bmod m + m) \bmod m$.

Dakle, određivanje modularnog multiplikativnog inverza broja možemo svesti na prošireni Euklidov algoritam.

```
// racunanje multiplikativnog inverza broja a po modulu m
// koriscenjem prosirenog Euklidovog algoritma
int modInverz(int a, int m){
  int x, y;
  // određujemo x i y tako da vazi a*x + m*y = d
  int d = nzdProsireni(a,m,x,y);

  // ako a i m nisu uzajamno prosti, onda ne postoji inverz
  if (d != 1)
    cout << "Modularni multiplikativni inverz ne postoji" << endl;
  else{
    // obezbedjujemo da je inverz iz skupa [0,m)
    int inverz = (x % m + m) % m;
    return inverz;
   }
}
```

Primetimo da za računanje modularnog multiplikativnog inverza broja vrednost koeficijenta y nije od interesa, te se može konstruisati algoritam koji ne poziva eksplicitno prošireni Euklidov algoritam, već na isti način kao kod proširenog Euklidovog algoritma iterativno računa samo vrednost koeficijenta x.

```
// funkcija racuna x -- multiplikativni inverz broja a po modulu m,
// koriscenjem prosirenog Euklidovog algoritma
// funkcija vraca da li je uspela da izracuna broj x
bool modInverz(int a, int m, int& x) {
  // pocetne vrednosti za r su m i a
  int r_preth = m;
  int r_tek = a;
  // pocetne vrednosti za x su 0 i 1
  int x_preth = 0;
  int x_tek = 1;

  while (r_tek > 0) {
    int q = r_preth / r_tek;
    int pom;

    // azuriramo tekucu i prethodnu vrednost niza r
    pom = r_preth;
    r_preth = r_tek;
    r_tek = pom - q * r_tek;

    // azuriramo tekucu i prethodnu vrednost niza x
    pom = x_preth;
```

```
    x_preth = x_tek;
    x_tek = pom - q * x_tek;
  }

  // obezbedjujemo da je inverz iz skupa [0,m)
  x = (x_preth % m + m) % m;

  // vracamo true ako je nzd(a,m) = 1, inace false
  return r == 1;
}
```

Složenost prethodnog algoritma za izračunavanje multiplikativnog inverza broja a po modulu m jednaka je složenosti proširenog Euklidovog algoritma za brojeve a i m i iznosi $O(\log(a+m))$, odnosno kada je $a < m$ jednaka je $O(\log m)$.

Algoritam zasnovan na Ojlerovoj i maloj Fermaovoj teoremi

Drugi način za računanje modularnog multiplikativnog inverza broja je korišćenjem Ojlerove teoreme. Podsetimo se njenog tvrđenja: ako su a i m uzajamno prosti brojevi onda važi:

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

Primetimo da je uslov da su brojevi a i m uzajamno prosti takođe uslov i da bi postojao multiplikativni inverz broja a po modulu m. Iz prethodne jednačine sledi:

$$a \cdot a^{\phi(m)-1} \equiv 1 \pmod{m}. \tag{2}$$

Primetimo da za proizvoljnu vrednost m važi $\phi(m) \geq 1$, odnosno $\phi(m) - 1 \geq 0$, te je $a^{\phi(m)-1}$ celobrojna vrednost. Dakle, za $x = a^{\phi(m)-1} \mod m$ važi $a \cdot x \equiv 1 \pmod{m}$, pa je $a^{\phi(m)-1}$ multiplikativni inverz broja a po modulu m. Dodatno, $a^{\phi(m)-1}$ može biti veće od m te je kao rezultat potrebno vratiti $x \mod m$.

Na ovaj način smo problem računanja modularnog multiplikativnog inverza broja a sveli na računanje Ojlerove funkcije broja m što se dalje svodi na faktorizaciju broja m, koja je složenosti $O(\sqrt{m})$.

Primer: Vratimo se na primer sa početka ovog poglavlja: za $m = 8$ i $a = 5$, važi $\phi(8) = 4$ i stoga važi $5^4 \equiv 1 \pmod{8}$. Odatle dobijamo $5 \cdot 5^3 \equiv 1 \pmod{8}$, tj. broj $5^3 \bmod 8 = 125 \bmod 8 = 5$ je multiplikativni inverz broja 5 po modulu 8.

Ukoliko je pak broj m prost, tvrđenje Ojlerove teoreme se svodi na malu Fermaovu teoremu:

$$a^{m-1} \equiv 1 \pmod{m}$$

odakle sledi:

$$a \cdot a^{m-2} \equiv 1 \pmod{m}. \tag{3}$$

Pošto je broj m prost, važi $m \geq 2$, odnosno $m - 2 \geq 0$, te je vrednost a^{m-2} celobrojna. Stoga važi da je $x = a^{m-2} \mod m$ multiplikativni inverz broja a po

modulu m. Primetimo da je ovde stvar jednostavnija, jer nam ne treba vrednost Ojlerove funkcije broja m.

Primer: Ako je $m = 7$ i $a = 3$, pošto je broj m prost, multiplikativni inverz broja 3 po modulu 7 možemo naći po formuli $a^{m-2} \bmod m = 3^5 \bmod 7 = 243 \bmod 7 = 5$ i zaista važi $3 \cdot 5 \equiv 1 \pmod 7$.

Iz kongruencija (2) i (3) možemo jednostavno odrediti multiplikativni inverz broja a po modulu m. Jedino što preostaje jeste što efikasnije izračunati odgovarajući stepen broja a, što se može izvesti korišćenjem efikasnog algoritma za stepenovanje broja, koji je složenosti $O(\log k)$, gde je k vrednost eksponenta. Naime, vrednost a^k možemo izračunati na sledeći način:

- kada je $k = 0$ važi $a^k = 1$;
- kada je $k \neq 0$
  - ako je k parno važi $a^k = (a^{k/2})^2$;
  - ako je k neparno, onda je $k - 1$ parno i važi $a^k = a \cdot (a^{k-1/2})^2$.

Primer: Ilustrujmo proceduru računanja trinaestog stepena broja a:

$$a^{13} = a \cdot a^{12} = a \cdot (a^6)^2 = a \cdot ((a^3)^2)^2 = a \cdot ((a \cdot a^2)^2)^2$$

Napomenimo da je u situaciji kada m nije prost broj, potrebno izračunati vrednost Ojlerove funkcije broja m što uključuje faktorizaciju broja m i može biti težak problem. Međutim, u situaciji kada je broj m prost, ali takođe i kada broj m nije prost ali je poznata njegova faktorizacija, složenost ovog algoritma je $O(\log m)$.

U nastavku je dat algoritam za računanje multiplikativnog inverza broja a po modulu m za prost broj m.

```
// funkcija za mnozenje po modulu m
int puta_mod(int a, int b, int m){
  return ((a % m) * (b % m)) % m;
}

// funkcija za brzo stepenovanje po modulu m
int stepen_mod(int a, int b, int m) {
    // a^0 = 1
    if (b == 0)
        return 1;
    // racunamo a^(b/2) mod m
    int rez = stepen_mod(a, b / 2, m);
    // racunamo rez*rez mod m
    rez = puta_mod(rez, rez, m);
    if (b % 2)
      // racunamo rez*a mod m
      return puta_mod(rez, a, m);
    else
```

```
        return rez;
}

// racunanje multiplikativnog inverza broja a po modulu m
// koriscenjem male Fermaove teoreme
int modInverz(int a, int m){
  return stepen_mod(a, m - 2, m);
}
```

Modularni multiplikativni inverz ima primenu u oblasti kriptografije, na primer u algoritmu RSA.

Algoritam RSA jedan je od prvih sistema šifrovanja koji se zasniva na javnom ključu. Naziv sistema predstavlja akronim prezimena njegovih autora (Rivest, Shamir, Adleman). U RSA algoritmu ključ koji se koristi za šifrovanje je javni i on se razlikuje od ključa koji se koristi za dešifrovanje koji je tajni. Razmotrimo kako izgledaju procedure šifrovanja i dešifrovanja poruka u algoritmu RSA. Koraci 1–6 odgovaraju fazi pripreme, dok koraci 7 i 8 odgovaraju redom koracima šifrovanja i dešifrovanja.

1. Izaberemo dva različita prosta broja p i q, koji su istog reda veličine.
2. Izračunamo $n = p \cdot q$.
3. Izračunamo vrednost Ojlerove funkcije broja n: $k = \phi(n) = (p-1) \cdot (q-1)$.
4. Biramo vrednost e koja je manja od $\phi(n)$ tako da je uzajamno prosta sa k.
5. Odredimo d kao multiplikativni inverz broja e po modulu k, odnosno tako da važi $e \cdot d \equiv 1 \pmod{k}$.
6. Brojevi n i e su javni, a broj d je privatni ključ.
7. Poruka m koja se predstavlja celim brojem manjim od n se šifruje računanjem vrednosti $S = m^e \bmod n$.
8. Poruka se dešifruje računanjem vrednosti $t = S^d \bmod n$.

Dokažimo da je $t = m$.

Razmotrimo slučaj kada je $\text{nzd}(m, n) = 1$. Tada je prema Ojlerovoj teoremi

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

te važi:

$$t = S^d = m^{ed} = m^{kl+1} = m \cdot m^{kl} = m \cdot m^{\phi(n)l} \equiv m \cdot 1 = m \pmod{n}.$$

Lako se proverava da je $t = m$ i ako uslov $\text{nzd}(m, n) = 1$ nije ispunjen.

Primer: Ilustrujmo prethodni postupak na jednom primeru. Neka je $p = 61$ i $q = 53$. Onda je $n = p \cdot q = 3233$ i važi $k = \phi(n) = (p-1) \cdot (q-1) = 60 \cdot 52 = 780$. Za broj e biramo broj manji od 780 koji je uzajamno prost sa 780, na primer $e = 17$. Modularni multiplikativni inverz broja 17 po modulu 780 jednak je $d = 413$. Šifrovanje poruke m svodi se na računanje

$$S = m^{17} \bmod 3233$$

a dešifrovanje poruke S na računanje

$$t = S^{413} \bmod 3233.$$

Na primer, šifrovanjem poruke $m = 65$ dobija se $S = 65^{17} \bmod 3233 = 2790$, a dešifrovanjem ove poruke dobijamo $t = 2790^{413} \bmod 3233 = 65$.

Sigurnost sistema RSA bi se narušila ako bi se broj n mogao efikasno faktorisati ili ako bi se $\phi(n)$ moglo izračunati bez faktorizacije broja n, na neki efikasniji način.

Kineska teorema o ostacima

Prema jednoj kineskoj legendi, general Han Xin je, da bi izbegao da špijuni saznaju sa kolikom je vojskom krenuo u boj, svom kuriru davao samo informaciju o ostatku broja vojnika u pravougaonoj formaciji. Njegove poruke su bile u narednoj formi: "Kada se vojnici organizuju u redove od po 9, preostaje njih 4; ako su u redovima od po 11, niko ne preostaje; ako su u redovima od po 13, samo jedan preostaje, a ako su u redovima od po 19, preostaje njih trojica". General je takođe svom kuriru preneo da je broj vojnika drugo najmanje rešenje ovog problema. Pitanje koje se postavlja je kolikim brojem vojnika je general zapovedao. Ovaj problem odgovara tzv. *kineskoj teoremi o ostacima*, jednom od standardnih problema elementarne teorije brojeva. Formulišimo ovaj problem u standardnim terminima.

Problem: Data su dva niza brojeva r: $r_0, r_1, \ldots, r_{n-1}$ i m: $m_0, m_1, \ldots, m_{n-1}$ pri čemu za svaki par brojeva niza m važi da su uzajamno prosti. Odrediti najmanji pozitivan broj x za koji važi:

$$\begin{aligned} x \bmod m_0 &= r_0 \\ x \bmod m_1 &= r_1 \\ &\ldots \\ x \bmod m_{n-1} &= r_{n-1} \end{aligned} \tag{4}$$

Kineska teorema o ostacima tvrdi da uvek postoji broj x koji zadovoljava ovaj skup jednačina i da je on jedinstven po modulu $m_0 \cdot m_1 \cdot \ldots \cdot m_{n-1}$.

Naivni pristup rešavanju ovog problema bi razmatrao redom brojeve od 1 naviše po skupu prirodnih brojeva i proveravao da li tekući broj zadovoljava date uslove. Pošto uvek postoji broj koji zadovoljava ovaj skup jednačina, na ovaj način bismo došli do rešenja, ali u broju koraka proporcionalnom traženoj vrednosti x.

```
int izracunajMinX(vector<int> m, vector<int> r){
  int n = m.size();
  // inicijalizujemo vrednost x
  int x = 1;

  // prema tvrđenju Kineske teoreme o ostacima
```

```
  // ova petlja ce se uvek zaustaviti
  while (true){
    int j;
    // za svako i od 0 do n-1 proveravamo
    // da li je ostatak pri deljenju broja x sa m_i jednak r_i
    for (i = 0; i < n; i++)
      if (x % m[i] != r[i])
        break;

    // ako su sve iteracije prosle uspesno
    // nasli smo vrednost za x
    if (i == n)
      return x;
    // inace nastavljamo sa pretragom
    x++;
  }
  return x;
}

int main(void){
  vector<int> m = {3, 4, 5};
  vector<int> r = {2, 3, 1};

  // m i r moraju biti istih dimenzija
  if (m.size() != r.size()){
    cout << "Broj modula mora biti jednak broju ostataka" << endl;
  }
  else
    cout << "Najmanje x koje zadovoljava dati sistem kongruencija je "
         << izracunajMinX(m, r) << endl;
  return 0;
}
```

Dokažimo tvrđenje kineske teoreme o ostacima.

Neka je $M = m_0 \cdot m_1 \cdot \ldots \cdot m_{n-1}$. Ako bismo umeli da odredimo cele brojeve $w_0, w_1, \ldots, w_{n-1}$ takve da za svako i važi da w_i pri deljenju sa m_i daje ostatak 1, dok pri deljenju sa svim drugim brojevima m_j, $j \neq i$ daje ostatak 0, onda bi se traženo x moglo konstruisati kao $x = r_0 \cdot w_0 + \ldots + r_{n-1} \cdot w_{n-1} \bmod M$.

	vrednost po mod m_0	vrednost po mod m_1	$\ldots$	vrednost po mod m_{n-1}
w_0	1	0	$\ldots$	0
w_1	0	1	$\ldots$	0
$\ldots$	$\ldots$	$\ldots$	$\ldots$	$\ldots$
w_{n-1}	0	0	$\ldots$	1

Pokazaćemo sada na koji način se mogu konstruisati brojevi w_i. Neka je:

$$z_0 = \frac{M}{m_0},\ z_1 = \frac{M}{m_1}\ , \ldots,\ z_{n-1} = \frac{M}{m_{n-1}}$$

Za svako $0 \leq i < n$ važi sledeće:

1. $z_i \equiv 0 \pmod{m_j}$ za $j \neq i$;
2. z_i i m_i su uzajamno prosti brojevi.

Ako je broj w_i deljiv svim brojevima $m_j, 0 \leq j < n$, osim sa m_i, onda je on deljiv i sa z_i (jer su svi m_i uzajamno prosti), odnosno mora da predstavlja neki njegov umnožak.

Uvedimo, dalje, naredne oznake:

$$\begin{aligned} y_0 &\equiv z_0^{-1} \pmod{m_0} \\ y_1 &\equiv z_1^{-1} \pmod{m_1} \\ &\ldots \\ y_{n-1} &\equiv z_{n-1}^{-1} \pmod{m_{n-1}} \end{aligned}$$

Ovi modularni multiplikativni inverzi postoje jer su brojevi z_i i m_i uzajamno prosti, i možemo ih odrediti, na primer, proširenim Euklidovim algoritmom. Primetimo da važi:

1. $y_i \cdot z_i \equiv 0 \pmod{m_j}$ za $j \neq i$;
2. $y_i \cdot z_i \equiv 1 \pmod{m_i}$.

Ova svojstva nam omogućavaju da definišemo brojeve $w_0, w_1, \ldots, w_{n-1}$ koji zadovoljavaju zahteve date u prethodnoj tabeli na sledeći način:

$$\begin{aligned} w_0 &= y_0 \cdot z_0 \bmod M \\ w_1 &= y_1 \cdot z_1 \bmod M \\ &\ldots \\ w_{n-1} &= y_{n-1} \cdot z_{n-1} \bmod M \end{aligned}$$

Neposredno se proverava da broj

$$x = r_0 \cdot w_0 + \ldots + r_{n-1} \cdot w_{n-1} \bmod M$$

daje ostatak r_i pri deljenju sa m_i. Naime,

$$\begin{aligned} x &\equiv r_0 y_0 z_0 + r_1 y_1 z_1 + \ldots + r_{n-1} y_{n-1} z_{n-1} \pmod{M} \pmod{m_i} \\ &\equiv r_0 y_0 z_0 + r_1 y_1 z_1 + \ldots + r_{n-1} y_{n-1} z_{n-1} \pmod{m_i} \\ &\equiv r_i y_i z_i \pmod{m_i} \\ &\equiv r_i \pmod{m_i} \end{aligned} \tag{5}$$

Druga jednakost u ovom nizu važi na osnovu toga što je $M \bmod m_i = 0$, treća na osnovu toga što je $y_j \cdot z_j \equiv 0 \pmod{m_i}$ za $j \neq i$, a četvrta na osnovu svojstva $y_i \cdot z_i \equiv 1 \pmod{m_i}$.

Pokažimo i jedinstvenost rešenja po modulu M. Pretpostavimo da u intervalu $[0, M)$ postoje dva različita rešenja u i v sistema jednačina (4). Tada važi:

$$m_0|(u-v),\ m_1|(u-v), \ldots,\ m_{n-1}|(u-v)$$

S obzirom na to da su svi $m_0, m_1, \ldots, m_{n-1}$ uzajamno prosti, važi i da

$$m_0 \cdot m_1 \cdot \ldots \cdot m_{n-1}|(u-v),$$

odnosno važi da je

$$u \equiv v \pmod{m_0 \cdot m_1 \cdot \ldots \cdot m_{n-1}},$$

odnosno rešenje je jedinstveno po modulu $m_0 \cdot m_1 \cdot \ldots \cdot m_{n-1}$.

Time je dokazana kineska teorema o ostacima. Na ovoj ideji zasniva se i efikasniji algoritam za rešavanje polaznog problema.

Primer: Vratimo se na polazni primer i izračunajmo kolikom je vojskom general Han Xin rukovodio. Ako sa x označimo broj vojnika, onda zadate uslove možemo da zapišemo na sledeći način:

$$\begin{aligned} x &\equiv 4 \pmod{9} \\ x &\equiv 0 \pmod{11} \\ x &\equiv 1 \pmod{13} \\ x &\equiv 3 \pmod{19} \end{aligned}$$

Označimo sa $M = 9 \cdot 11 \cdot 13 \cdot 19 = 24453$ i sa:

$$z_0 = \frac{M}{9} = 2717,\ z_1 = \frac{M}{11} = 2223,\ z_2 = \frac{M}{13} = 1881,\ z_3 = \frac{M}{19} = 1287.$$

Izračunajmo modularne multiplikativne inverze ovih brojeva:

$$\begin{aligned} y_0 &= z_0^{-1} = 2717^{-1} \bmod 9 = 8^{-1} \bmod 9 = 8 \\ y_1 &= z_1^{-1} = 2223^{-1} \bmod 11 = 1^{-1} \bmod 11 = 1 \\ y_2 &= z_2^{-1} = 1881^{-1} \bmod 13 = 9^{-1} \bmod 13 = 3 \\ y_3 &= z_3^{-1} = 1287^{-1} \bmod 19 = 14^{-1} \bmod 19 = 15 \end{aligned}$$

Na osnovu ovih vrednosti, možemo izračunati potrebne brojeve w_i:

$$\begin{aligned} w_0 &= y_0 \cdot z_0 \bmod M = 8 \cdot 2717 \bmod 24453 = 21736 \\ w_1 &= y_1 \cdot z_1 \bmod M = 1 \cdot 2223 \bmod 24453 = 2223 \\ w_2 &= y_2 \cdot z_2 \bmod M = 3 \cdot 1881 \bmod 24453 = 5643 \\ w_3 &= y_3 \cdot z_3 \bmod M = 15 \cdot 1287 \bmod 24453 = 19305 \end{aligned}$$

Napokon dobijamo:

$$\begin{aligned} x &\equiv r_0 \cdot w_0 + \ldots + r_{n-1} \cdot w_{n-1} \pmod{M} \\ &\equiv 4 \cdot 21736 + 0 \cdot 2223 + 1 \cdot 5643 + 3 \cdot 19305 \pmod{24453} \\ &\equiv 3784 \pmod{24453} \end{aligned}$$

S obzirom na to da je rečeno da je broj vojnika drugo najmanje rešenje ovog problema, ukupan broj vojnika jednak je

$$x = 3784 + 24453 = 28237$$

Razmotrimo sada implementaciju rešenja polaznog problema zasnovanog na tvrđenju kineske teoreme o ostacima.

```
// proizvod brojeva x i y po modulu m
long long pm(long long x, long long y, long long m) {
  return ((x % m) * (y % m)) % m;
}

// proizvod brojeva x, y i z po modulu m
// dva puta se poziva funkcija za racunanje proizvoda po modulu m
long long pm(long long x, long long y, long long z, long long m) {
  return pm(pm(x, y, m), z, m);
}

// zbir brojeva x i y po modulu m
long long zm(long long x, long long y, long long m) {
  return ((x % m) + (y % m)) % m;
}

// na osnovu kineske teoreme o ostacima se izracunava rezultat tako da
// za sve elemente nizova m i a vazi da je rezultat mod m[i] = r[i]
// funkcija vraca da li je rezultat bilo moguce naci
bool kto(long long m[], long long r[], int n, long long& rezultat) {

  long long M = 1;
  // racunamo proizvod svih modula
  for (int i = 0; i < n; i++)
    M *= m[i];

  rezultat = 0;
  for (int i = 0; i < n; i++) {
    // racunamo zi
    long long zi = M / m[i];
    long long yi;
    // racunamo yi kao inverz broja zi po modulu m[i]
```

```
    if (!modInverz(zi, m[i], yi))
      return false;
    // na rezultat dodajemo sabirak ri*yi*zi mod M
    rezultat = zm(rezultat, pm(r[i], yi, zi, M), M);
  }
  return true;
}

int main(){
  // ucitavamo ulazne podatke
  long long r[3], m[3];
  for (int i = 0; i < 3; i++)
    cin >> r[i] >> m[i];

  // rezultat izracunavamo na osnovu kineske teoreme o ostacima
  long long rezultat;
  if (kto(m, r, 3, rezultat))
    // ispisujemo rezultat
    cout << rezultat << endl;
  return 0;
}
```

Postoji još jedan postupak koji je manje matematički zahtevan nego prethodni. Pristup je zasnovan na pametnoj pretrazi. Ako broj x pri deljenju sa brojem m_0 daje ostatak r_0 onda je on sigurno jedan od članova aritmetičkog niza $r_0, r_0+m_0, r_0+2m_0, \ldots$ U petlji redom proveravamo ove brojeve sve dok ne naiđemo na prvi element koji pri deljenju sa m_1 daje ostatak r_1. Kada pronađemo takav broj r_{01} (on će biti najmanji pozitivan broj sa osobinom da pri deljenju sa m_0 i m_1 daje redom ostatke r_0 i r_1), znamo da će svaki naredni broj koji zadovoljava to svojstvo biti član aritmetičkog niza r_{01}, $r_{01}+m_0 \cdot m_1$, $r_{01}+2 \cdot m_0 \cdot m_1, \ldots$ Redom pretražujemo elemente ovog niza dok ne naiđemo na element r_{012} koji pri deljenju sa m_2 daje ostatak r_2, a zatim posmatramo brojeve r_{012}, $r_{012} + m_0 \cdot m_1 \cdot m_2$, $r_{012} + 2 \cdot m_0 \cdot m_1 \cdot m_2$ i tako dalje.

Brza Furijeova transformacija

Brza Furijeova transformacija (ili FFT, što je skraćenica od eng. Fast Fourier transform) je važna iz velikog broja razloga. Ona efikasno rešava važan praktičan problem, elegantna je, i otvara nove, neočekivane oblasti primene. Zbog toga ona predstavlja jedan od najvažnijih algoritama od svog otkrića sredinom šezdesetih godina dvadesetog veka i nalazi se na spisku deset najznačnijih algoritama dvadesetog veka po IEEE-u (https://www.andrew.cmu.edu/course/15-355/misc/Top%20Ten%20Algorithms.html). Tačnije, brza Furijeova transformacija je dobila na popularnosti radom Kulija i Tukija 1965. godine, međutim, kasnije se otkrilo da su ova dva autora nezavisno iznova "izmislili" algoritam koji je osmislio Gaus još davne 1805. godine.

Brza Furijeova transformacija ima brojne primene u inženjerstvu, u obradi digitalnih signala poput obrade zvuka i obrade digitalnih slika i u matematici. Mi ćemo se u ovom materijalu ograničiti na jednu njenu primenu, množenje polinoma.

Problem: Izračunati proizvod dva zadata polinoma $P(x)$ i $Q(x)$.

Formulacija problema koji treba rešiti je precizna samo na prvi pogled, jer nije preciziran način na koji su polinomi predstavljeni. Obično se polinom:

$$P(x) = a_{n-1}x^{n-1} + a_{n-2}x^{n-2} + \ldots + a_1x + a_0$$

predstavlja nizom svojih koeficijenata uz $1, x, x^2, \ldots, x^{n-1}$; međutim, to nije jedina mogućnost. Alternativno, polinom stepena $n-1$ moguće je predstaviti njegovim vrednostima u n različitih tačaka: te vrednosti jednoznačno određuju polinom. Na primer, polinom stepena 1, odnosno linearna funkcija, jedinstveno je određena vrednostima u dve različite tačke, a polinom stepena 2, odnosno kvadratna funkcija, vrednostima u tri različite tačke.

Množenje dva polinoma koja su zadata nizom svojih koeficijenata može se izvršiti osnovnim algoritmom složenosti $O(n^2)$ koji se sastoji u množenju svakog monoma prvog polinoma svakim monomom drugog, ili korišćenjem Karacubinog algoritma zasnovanog na pametnoj primeni dekompozicije koji je složenosti $O(n^{\log_2 3})$.

Predstavljanje polinoma vrednostima na skupu tačaka je interesantno zbog jednostavnosti množenja. Naime, proizvod dva polinoma stepena $n-1$ je polinom stepena $2n-2$, pa je određen svojim vrednostima u $2n-1$ tačaka. Ako pretpostavimo da su vrednosti polinoma – činilaca stepena $n-1$ date u $2n-1$ različitih tačaka, onda se proizvod ova dva polinoma može izračunati pomoću $2n-1$, odnosno $O(n)$ običnih množenja. Naime, vrednost polinoma PQ u tački a jednaka je proizvodu vrednosti polinoma P u tački a i vrednosti polinoma Q u tački a.

Nažalost, predstavljanje polinoma njegovim vrednostima na skupu tačaka nije pogodno za neke primene. Primer je izračunavanje vrednosti polinoma u proizvoljnoj tački; pri reprezentaciji polinoma vrednostima na skupu tačaka, ovo je mnogo teže (u pitanju je problem interpolacije) u odnosu na to kada je polinom zadat nizom svojih koeficijenata. Međutim, ako bismo mogli da efikasno prevodimo polinome iz jedne u drugu reprezentaciju, dobili bismo odličan algoritam za množenje polinoma. Upravo to se postiže primenom algoritma FFT.

Tabela 2: Složenost osnovnih operacija za rad sa polinomima u zavisnosti od reprezentacije polinoma.

reprezentacija	računanje vrednosti polinoma u tački	množenje polinoma
koeficijenti	$\mathbf{O(n)}$	$O(n^2)$ ili $O(n^{\log_2 3})$
vrednosti	$O(n^2)$	$\mathbf{O(n)}$

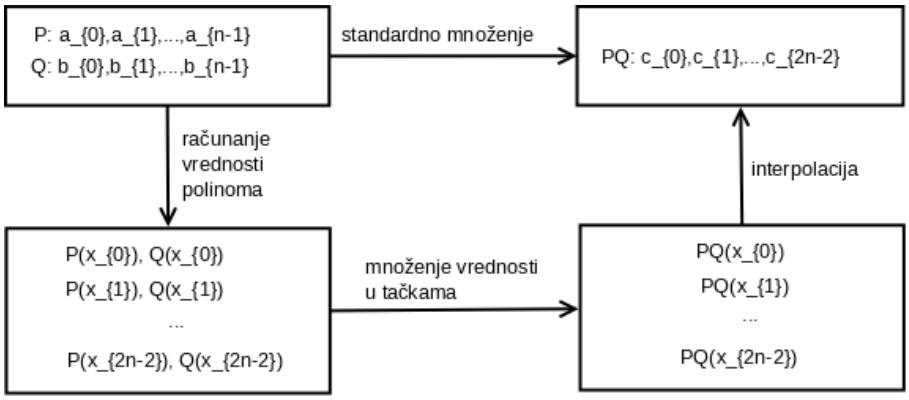
Prelaz od predstave polinoma koeficijentima na predstavu vrednostima u tačkama, rešava se izračunavanjem vrednosti polinoma u tim tačkama. Vrednost polinoma $P(x)$ (zadatog koeficijentima) u proizvoljnoj tački x_0 može se pomoću Hornerove šeme izračunati korišćenjem n množenja:

$$P(x_0) = a_0 + x_0(a_1 + x_0(a_2 + \ldots + x_0 \cdot a_{n-1} \ldots))$$

Izračunavanje vrednosti polinoma $P(x)$ u n proizvoljnih tačaka izvodljivo je dakle pomoću n^2 množenja. Prelaz od predstave polinoma vrednostima na skupu tačaka na predstavu koeficijentima ćemo u nastavku takođe zvati interpolacijom. Vrednosti koeficijenta polinoma na osnovu vrednosti $(x_j, y_j), 0 \leq j < n$ polinoma u n različitih tačaka mogu se odrediti korišćenjem Lagranžove interpolacione formule:

$$P(x) = \sum_{j=0}^{n-1} a_j x^j = \sum_{j=0}^{n-1} y_j \cdot \frac{\prod_{k \neq j}(x - x_k)}{\prod_{k \neq j}(x_j - x_k)}$$

Na osnovu ovog izraza koeficijenti polinoma mogu se odrediti algoritmom složenosti $O(n^2)$. Dakle, imamo sledeću situaciju: u svakoj od razmatranih reprezentacija jedna operacija se izvršava efikasno, a druga ne (tabela 2).



Slika 1: Dve mogućnosti za računanje proizvoda dva polinoma $P(x) = \sum_{i=0}^{n-1} a_i x^i$ i $Q(x) = \sum_{i=0}^{n-1} b_i x^i$: direktnim množenjem polinoma što je složenosti $\Theta(n^2)$ (gornja strelica udesno) ili preko reprezentacije polinoma vrednostima na skupu tačaka (strelica nadole, donja strelica udesno, strelica nagore).

Ovde je ključna ideja da se ne koristi proizvoljnih n tačaka: mi imamo slobodu da po želji izaberemo pogodan skup od n različitih tačaka. Brza Furijeova transformacija koristi specijalan skup tačaka, tako da se obe operacije, i izračunavanje vrednosti polinoma na skupu tačaka i interpolacija, mogu efikasnije izvršavati.

Direktna Furijeova transformacija

Razmotrimo problem izračunavanja vrednosti polinoma. Potrebno je izračunati vrednosti dva polinoma stepena $n-1$ u $2n-1$ tačaka, da bi se njihov proizvod, polinom stepena $2n-2$, mogao interpolirati. Međutim, polinom stepena $n-1$ može se predstaviti kao polinom stepena $2n-2$ izjednačavanjem sa nulom vodećih $n-1$ koeficijenata:

$$\begin{aligned} P_{n-1}(x) &= a_{n-1}x^{n-1} + \ldots + a_1 x + a_0 = \\ P_{2n-2}(x) &= 0 \cdot x^{2n-2} + \ldots + 0 \cdot x^n + a_{n-1}x^{n-1} + \ldots + a_1 x + a_0 \end{aligned}$$

Zbog toga se bez gubitka opštosti može pretpostaviti da je problem izračunati vrednosti proizvoljnog polinoma $P = \sum_{j=0}^{n-1} a_j x^j$ stepena $n-1$ u n različitih tačaka. Cilj je pronaći takvih n tačaka, u kojima je lako izračunati vrednosti polinoma. Zbog jednostavnosti pretpostavljamo da je n stepen dvojke.

Koristićemo matričnu terminologiju da bismo uprostili označavanje. Izračunavanje vrednosti polinoma $P = \sum_{j=0}^{n-1} a_j x^j$ u n tačaka $x_0, x_1, \ldots, x_{n-1}$ može se predstaviti kao izračunavanje proizvoda naredne matrice i vektora:

$$\begin{pmatrix} 1 & x_0 & x_0^2 & \ldots & x_0^{n-1} \\ 1 & x_1 & x_1^2 & \ldots & x_1^{n-1} \\ \ldots & \ldots & \ldots & \ldots & \ldots \\ 1 & x_{n-1} & x_{n-1}^2 & \ldots & x_{n-1}^{n-1} \end{pmatrix} \begin{pmatrix} a_0 \\ a_1 \\ \ldots \\ a_{n-1} \end{pmatrix} = \begin{pmatrix} P(x_0) \\ P(x_1) \\ \ldots \\ P(x_{n-1}) \end{pmatrix} \tag{6}$$

Matricu iz jednačine (6) koja sadrži stepene brojeva x_i nazivamo *Vandermondova matrica*. Pitanje je kako izabrati vrednosti $x_0, x_1, \ldots, x_{n-1}$, tako da se ovo množenje što je više moguće uprosti. Posmatrajmo dve proizvoljne vrste r i s Vandermondove matrice. Voleli bismo da ih učinimo što sličnijim, da bismo uštedeli na množenjima. Ne može se staviti $x_r = x_s$, jer tačke moraju biti različite, ali se $x_r^2 = x_s^2$ može postići stavljajući $x_s = -x_r$. Ovo je dobar izbor, jer je svaki paran stepen broja x_r jednak odgovarajućem parnom stepenu broja x_s, dok se neparni stepeni razlikuju samo po znaku. Isto se može uraditi i sa ostalim parovima vrsta. Naslućuje se u kom pravcu treba tražiti n specijalnih vrsta, za koje bi se gornji proizvod svodio na samo $n/2$ proizvoda vrsta Vandermondove matrice sa kolonom koeficijenata. Rezultat bi bio polovljenje veličine ulaza, a time i vrlo efikasan algoritam. Pokušajmo da postavimo ovaj problem kao dva odvojena problema dvostruko manje veličine.

Podela polaznog problema na dva potproblema veličine $m = n/2$ može se opisati

sledećim izrazom

$$\begin{pmatrix} 1 & x_0 & x_0^2 & \dots & x_0^{n-1} \\ 1 & x_1 & x_1^2 & \dots & x_1^{n-1} \\ \dots & \dots & \dots & \dots & \dots \\ 1 & x_{m-1} & x_{m-1}^2 & \dots & x_{m-1}^{n-1} \\ 1 & -x_0 & (-x_0)^2 & \dots & (-x_0)^{n-1} \\ 1 & -x_1 & (-x_1)^2 & \dots & (-x_1)^{n-1} \\ \dots & \dots & \dots & \dots & \dots \\ 1 & -x_{m-1} & (-x_{m-1})^2 & \dots & (-x_{m-1})^{n-1} \end{pmatrix} \begin{pmatrix} a_0 \\ a_1 \\ \dots \\ a_{n-1} \end{pmatrix} = \begin{pmatrix} P(x_0) \\ P(x_1) \\ \dots \\ P(x_{m-1}) \\ P(-x_0) \\ \dots \\ P(-x_{m-1}) \end{pmatrix}. \tag{7}$$

Polazna $n \times n$ matrica podeljena je na dve vrlo slične podmatrice dimenzija $n/2 \times n$. Za svako $j = 0, 1, \dots, n/2 - 1$, važi $x_{n/2+j} = -x_j$. Zgodno je napisati izraze za $P(x_j)$ i $P(-x_j)$, odnosno uopšte $P(x)$, sa razdvojenim članovima parnog i neparnog stepena:

$$\begin{aligned} P(x) &= \sum_{j=0}^{n-1} a_j x^j \\ &= \sum_{j=0}^{n/2-1} a_{2j} x^{2j} + \sum_{j=0}^{n/2-1} a_{2j+1} x^{2j+1} \\ &= \sum_{j=0}^{n/2-1} a_{2j} x^{2j} + x \cdot \sum_{j=0}^{n/2-1} a_{2j+1} x^{2j} \end{aligned} \tag{8}$$

Ako sa $P_p(x)$ i $P_{np}(x)$ označimo polinome stepena $n/2 - 1$ sa koeficijentima polinoma $P(x)$ parnog, odnosno neparnog indeksa:

$$\begin{aligned} P_p(x) &= \sum_{j=0}^{n/2-1} a_{2j} x^j \\ P_{np}(x) &= \sum_{j=0}^{n/2-1} a_{2j+1} x^j \end{aligned} \tag{9}$$

dolazimo do veoma važne jednakosti:

$$P(x) = P_p(x^2) + x \cdot P_{np}(x^2). \tag{10}$$

Zamenom x sa $-x$ u prethodnoj jednačini dobijamo

$$P(-x) = P_p(x^2) + (-x) \cdot P_{np}(x^2).$$

Izračunavanje $P(x_j)$ za $j = 0, 1, \dots, n - 1$ svodi se na računanje $P(x_j)$ i $P(-x_j)$ za $j = 0, 1, \dots, n/2 - 1$, odnosno na izračunavanje samo $n/2$ vrednosti $P_p(x_j^2)$, $n/2$ vrednosti $P_{np}(x_j^2)$, i dopunskih $n/2$ sabiranja, $n/2$ oduzimanja i n množenja.

Dakle, problem dimenzije n sveli smo na dva potproblema dimenzije $n/2$ i $O(n)$ dopunskih operacija. Primetimo da je potrebno izračunati vrednosti polinoma P_p i P_{np} u kvadratima vrednosti polaznih tačaka, a njih ima $n/2$, odnosno duplo manje od broja polaznih tačaka.

Primer: Neka je dat polinom $P(x) = 3+2x+4x^2-5x^3$. Njega možemo zapisati kao

$$P(x) = (3 + 4x^2) + (2x - 5x^3) = (3 + 4x^2) + x(2 - 5x^2)$$

te važi $P_p(x) = 3 + 4x$ i $P_{np}(x) = 2 - 5x$. Važi:

$$\begin{aligned} P(x_0) &= (3 + 4x_0^2) + x_0(2 - 5x_0^2) = P_p(x_0^2) + x_0 \cdot P_{np}(x_0^2) \\ P(-x_0) &= (3 + 4x_0^2) - x_0(2 - 5x_0^2) = P_p(x_0^2) - x_0 \cdot P_{np}(x_0^2) \end{aligned}$$

Može li se nastaviti rekurzivno na isti način? Ako bi nam to pošlo za rukom, došli bismo do poznate diferencne jednačine $T(n) = 2T(n/2) + O(n)$, čije je rešenje $T(n) = O(n \log n)$. Problem izračunavanja $P(x)$ (polinoma stepena $n-1$) u n tačaka sveli smo na izračunavanje $P_p(x^2)$ i $P_{np}(x^2)$ (dva polinoma stepena $n/2-1$) u $n/2$ tačaka. To je regularna redukcija, izuzev jednog detalja: vrednosti x u $P(x)$ mogu se proizvoljno birati, ali vrednosti x^2 u izrazu (na primer) $P_p(x^2)$ mogu biti samo pozitivne. Pošto smo do redukcije došli korišćenjem negativnih brojeva, ovo predstavlja problem. Izdvojimo iz jednačine (7) matricu dimenzije $n/2 \times n/2$ koja odgovara izračunavanju vrednosti $P_p(x^2)$:

$$\begin{pmatrix} 1 & x_0^2 & x_0^4 & \dots & x_0^{n-2} \\ 1 & x_1^2 & x_1^4 & \dots & x_1^{n-2} \\ \dots & \dots & \dots & \dots & \dots \\ 1 & x_{n/2-1}^2 & x_{n/2-1}^4 & \dots & x_{n/2-1}^{n-2} \end{pmatrix} \begin{pmatrix} a_0 \\ a_2 \\ \dots \\ a_{n-2} \end{pmatrix} = \begin{pmatrix} P_p(x_0^2) \\ P_p(x_1^2) \\ \dots \\ P_p(x_{n/2-1}^2) \end{pmatrix}.$$

Da bismo još jednom izveli redukciju na isti način, morali bismo da stavimo, na primer, $x_{n/4}^2 = -(x_0^2)$. Pošto su kvadrati realnih brojeva uvek pozitivni, ovo nije moguće, bar ako se ograničimo na realne brojeve. Poteškoća se prevazilazi prelaskom na kompleksne brojeve. Problem se može opet podeliti na dva dela stavljajući $x_{j+n/4} = ix_j$, za $j = 0, 1, \dots, n/4 - 1$, gde je i koren iz -1, odnosno kompleksni broj. Ovo razdvajanje zadovoljava iste uslove kao i prethodno. Prema tome, problem veličine $n/2$ može se rešiti svođenjem na dva potproblema veličine $n/4$, izvodeći $O(n)$ dopunskih operacija.

Za sledeću redukciju potreban nam je broj z takav da je $z^8 = 1$ i $z^j \neq 1$ za $0 < j < 8$, odnosno *primitivni osmi koren iz jedinice*; tada je $z^4 = -1$ i $z^2 = i$. U opštem slučaju, potreban nam je *primitivni n-ti koren iz jedinice* (eng. principal n-th root of unity). Označimo ga sa ω (zbog jednostavnosti se n ne spominje eksplicitno; u okviru ovog odeljka radi se uvek o jednom istom n). Primitivni n-ti koren iz jedinice ω zadovoljava sledeće uslove:

$$\omega^n = 1,\ \omega^j \neq 1 \text{ za } 0 < j < n. \tag{11}$$

Na primer, drugi primitivni koren iz jedinice je -1, a četvrti primitivni koren iz jedinice imaginarna jedinica i.

Specijalno, prema Ojlerovoj formuli koja uspostavlja vezu između trigonometrijskih funkcija i kompleksne eksponencijalne funkcije važi:

$$e^{ix} = \cos x + i \sin x$$

Specijalno za $x = \pi$ dobija se:

$$e^{i\pi} = -1,$$

odnosno

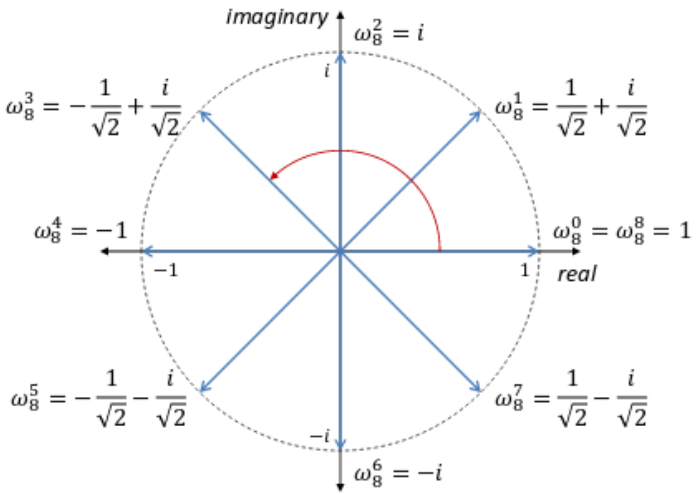
$$e^{2i\pi} = 1.$$

Odavde sledi da je vrednost:

$$\omega = e^{2\pi i/n} = \cos \frac{2\pi}{n} + i \sin \frac{2\pi}{n}$$

primitivni n-ti koren iz jedinice. U opštem slučaju treba da važi $x_{i+1} = \omega \cdot x_i$, a ako krenemo od $x_0 = 1$, za n tačaka $x_0, x_1, \ldots, x_{n-1}$ u kojima računamo vrednost polinoma dobijamo brojeve $1, \omega, \omega^2, \ldots, \omega^{n-1}$. Važi sledeće (videti primer na slici 2 za $n = 8$):

$$\omega^k = e^{(2\pi i/n)k} = \cos \frac{2k\pi}{n} + i \sin \frac{2k\pi}{n}$$



Slika 2: Vizuelizacija osmog primitivnog korena iz jedinice i njegovih stepena u kompleksnoj ravni.

Prema tome, potrebno je izračunati sledeći proizvod:

$$\begin{pmatrix} 1 & 1 & 1 & \ldots & 1 \\ 1 & \omega & \omega^2 & \ldots & \omega^{n-1} \\ 1 & \omega^2 & \omega^{2\cdot 2} & \ldots & \omega^{2\cdot(n-1)} \\ \ldots & \ldots & \ldots & \ldots & \ldots \\ 1 & \omega^{n-1} & \omega^{(n-1)\cdot 2} & \ldots & \omega^{(n-1)\cdot(n-1)} \end{pmatrix} \begin{pmatrix} a_0 \\ a_1 \\ \ldots \\ a_{n-1} \end{pmatrix} = \begin{pmatrix} P(1) \\ P(\omega) \\ \ldots \\ P(\omega^{n-1}) \end{pmatrix}. \quad (12)$$

Ovaj proizvod se zove *diskretna Furijeova transformacija* (eng. Discrete Fourier Transform) vektora koeficijenata $(a_0, a_1, \ldots, a_{n-1})$.

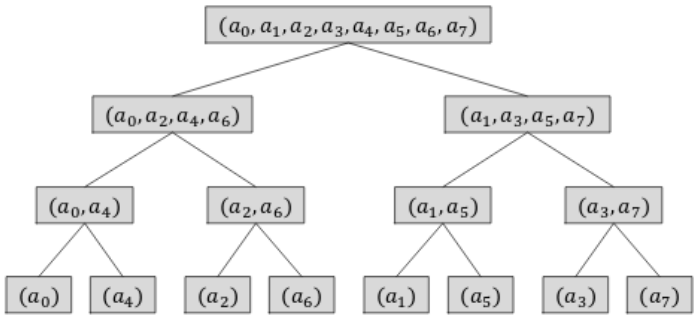
Zapazimo najpre da je ispunjen uslov

$$x_{n/2+j} = \omega^{n/2+j} = \omega^{n/2}\omega^j = -x_j, \quad j = 0, 1, \ldots, n/2-1$$

i na osnovu jednačine (10) važi:

$$\begin{aligned} P(x_j) &= P_p(x_j^2) + x_j P_{np}(x_j^2), \\ P(x_{n/2+j}) &= P_p(x_j^2) - x_j P_{np}(x_j^2). \end{aligned} \tag{13}$$

Prema tome, prva redukcija problema veličine n na dva manja je i dalje ispravna. Dalje, dva potproblema proizašla iz ove redukcije imaju po $n/2$ tačaka $1, \omega^2, \omega^4, \ldots, \omega^{n-2}$, što je upravo problem veličine $n/2$, u kome umesto ω figuriše ω^2 — primitivni $n/2$-ti koren iz jedinice. Da je ω^2 primitivni $n/2$-ti koren iz jedinice neposredno sledi iz uslova (11). Prema tome, dalje se može nastaviti rekurzivno (slika 3). Složenost algoritma zadovoljava diferencnu jednačinu $T(n) = 2T(n/2) + O(n)$, čije je rešenje $O(n \log n)$. Algoritam omogućuje efikasno izračunavanje Furijeove transformacije vektora koeficijenata polinoma, pa je dobio ime brza Furijeova transformacija, odnosno FFT.



Slika 3: Razlaganje ulaza problema FFT na potprobleme za $n = 8$.

Algoritam *FFT*
Ulaz: broj n koeficijenata polinoma P, koeficijenti polinoma $a_0, a_1, \ldots, a_{n-1}$, primitivni n-ti koren iz jedinice ω
Izlaz: vektor P koji sadrži vrednost polinoma P na skupu tačaka $1, \omega, \ldots, \omega^{n-1}$

```
1.  if n = 1
2.          P[0] ← a_0
3.      else
4.          P_p ← FFT(n/2, a_0, a_2, ..., a_{n-2}, w^2)
5.          P_np ← FFT(n/2, a_1, a_3, ..., a_{n-1}, w^2)
6.          for j ← 0 to n/2 − 1
```

7. $P[j] \leftarrow P_p[j] + w^j \cdot P_{np}[j]$
8. $P[j + n/2] \leftarrow P_p[j] - w^j \cdot P_{np}[j]$
9. **return** P

Primer: Ilustrujmo problem izračunavanja brze Furijeove transformacije vektora koeficijenata polinoma na primeru polinoma $3 + 2x + 5x^2 - 3x^3$ sa koeficijentima $(3, 2, 5, -3)$. Da bismo izbegli zabunu, potprobleme ćemo označavati sa $P_{j_0,j_1,\ldots,j_k}(x_0, x_1, \ldots, x_k)$, gde $j_0, j_1, \ldots, j_k$ označavaju koeficijente polinoma, a $x_0, x_1, \ldots, x_k$ tačke u kojima treba izračunati vrednosti polinoma. Zadatak je, dakle, rešiti problem $P_{3,2,5,-3}(1, \omega, \omega^2, \omega^3)$, tako da važi $\omega^4 = 1$, odnosno $w = i$ i to na osnovu jednačine (13).

Prvi korak je svođenje problema $P_{3,2,5,-3}(1, \omega, \omega^2, \omega^3)$ na potprobleme $P_{3,5}(1, \omega^2)$ i $P_{2,-3}(1, \omega^2)$. Nastavljamo rekurzivno na isti način i svodimo problem $P_{3,5}(1, \omega^2)$ na potprobleme $P_3(1)$ i $P_5(1)$. Polinomi $P_3(x)$ i $P_5(x)$ su konstantni polinomi i važi $P_3(1) = 3$ i $P_5(1) = 5$. Kombinovanjem ovih rezultata dobijamo

$$\begin{aligned} P_{3,5}(1) &= P_3(1) + 1 \cdot P_5(1) = 3 + 1 \cdot 5 = 8, \\ P_{3,5}(\omega^2) &= P_3(1) - 1 \cdot P_5(1) = 3 - 1 \cdot 5 = -2. \end{aligned}$$

Posle objedinjavanja ova dva rezultata dobijamo

$$P_{3,5}(1, \omega^2) = (8, -2).$$

Na isti način dobijamo

$$P_{2,-3}(1, \omega^2) = (-1, 5).$$

Kombinovanjem ova dva vektora dobijamo $P_{3,2,5,-3}(1, \omega, \omega^2, \omega^3)$:

$$\begin{aligned} P_{3,2,5,-3}(1) &= P_{3,5}(1) + 1 \cdot P_{2,-3}(1) = 8 + (-1) = 7, \\ P_{3,2,5,-3}(\omega) &= P_{3,5}(\omega^2) + \omega \cdot P_{2,-3}(\omega^2) = -2 + \omega \cdot 5 = -2 + 5i, \\ P_{3,2,5,-3}(\omega^2) &= P_{3,5}(1) - 1 \cdot P_{2,-3}(1) = 8 - (-1) = 9, \\ P_{3,2,5,-3}(\omega^3) &= P_{3,5}(\omega^2) - \omega \cdot P_{2,-3}(\omega^2) = -2 - \omega \cdot 5 = -2 - 5i, \end{aligned}$$

odnosno

$$P_{3,2,5,-3}(1, \omega, \omega^2, \omega^3) = (7, -2 + 5i, 9, -2 - 5i)).$$

Primer: Ilustrujmo sada problem izračunavanja brze Furijeove transformacije vektora na nešto složenijem primeru. Razmotrimo polinom $x + 2x^2 + \ldots + 7x^7$ sa koeficijentima $(0, 1, 2, 3, 4, 5, 6, 7)$. Zadatak je izračunati vrednost polinoma P na datom skupu tačaka, odnosno rešiti problem $P_{0,1,2,3,4,5,6,7}(1, \omega, \omega^2, \ldots, \omega^7)$.

Prvi korak je svođenje problema $P_{0,1,2,3,4,5,6,7}(1, \omega, \omega^2, \ldots, \omega^7)$ na potprobleme $P_{0,2,4,6}(1, \omega^2, \omega^4, \omega^6)$ i $P_{1,3,5,7}(1, \omega^2, \omega^4, \omega^6)$. Nastavljamo rekurzivno na isti način i svodimo problem $P_{0,2,4,6}(1, \omega^2, \omega^4, \omega^6)$ na potprobleme $P_{0,4}(1, \omega^4)$ i $P_{2,6}(1, \omega^4)$. Problem $P_{0,4}(1, \omega^4)$ se zatim svodi na potprobleme $P_0(1) = 0$ i $P_4(1) = 4$. Kombinovanjem ovih rezultata dobijamo

$$\begin{aligned} P_{0,4}(1) &= P_0(1) + 1 \cdot P_4(1) = 0 + 1 \cdot 4 = 4, \\ P_{0,4}(\omega^4) &= P_0(1) - 1 \cdot P_4(1) = 0 - 1 \cdot 4 = -4. \end{aligned}$$

Posle objedinjavanja ova dva rezultata dobijamo

$$P_{0,4}(1,\omega^4) = (4,-4).$$

Na isti način dobijamo

$$P_{2,6}(1,\omega^4) = (8,-4).$$

Sada kombinovanjem ova dva vektora dobijamo $P_{0,2,4,6}(1,\omega^2,\omega^4,\omega^6)$:

$$\begin{aligned}
P_{0,2,4,6}(1) &= P_{0,4}(1) + 1\cdot P_{2,6}(1) = 4+8 = 12,\\
P_{0,2,4,6}(\omega^2) &= P_{0,4}(\omega^4) + \omega^2\cdot P_{2,6}(\omega^4) = -4+\omega^2(-4),\\
P_{0,2,4,6}(\omega^4) &= P_{0,4}(1) - 1\cdot P_{2,6}(1) = 4-8 = -4,\\
P_{0,2,4,6}(\omega^6) &= P_{0,4}(\omega^4) - \omega^2\cdot P_{2,6}(\omega^4) = -4-\omega^2(-4),
\end{aligned}$$

odnosno

$$P_{0,2,4,6}(1,\omega^2,\omega^4,\omega^6) = (12,-4(1+\omega^2),-4,-4(1-\omega^2)).$$

Na sličan način dobija se

$$P_{1,3,5,7}(1,\omega^2,\omega^4,\omega^6) = (16,-4(1+\omega^2),-4,-4(1-\omega^2)).$$

Preostaje još da se izračuna osam vrednosti $P_{0,1,2,3,4,5,6,7}(1,\omega,\omega^2,\ldots,\omega^7)$. Važi:

$$\begin{aligned}
P_{0,1,2,3,4,5,6,7}(1) &= 12+1\cdot 16 = 28\\
P_{0,1,2,3,4,5,6,7}(\omega^4) &= 12-1\cdot 16 = -4
\end{aligned}$$

i slično:

$$\begin{aligned}
P_{0,1,2,3,4,5,6,7}(\omega) &= (-4(1+\omega^2)) + \omega\cdot(-4(1+\omega^2))\\
P_{0,1,2,3,4,5,6,7}(\omega^5) &= (-4(1+\omega^2)) - \omega\cdot(-4(1+\omega^2))
\end{aligned}$$

Na sličan način dobili bismo vrednosti polinoma P u tačkama $\omega^2,\omega^3,\omega^6,\omega^7$. Konačno dobijamo:

$$\begin{aligned}
&P_{0,1,2,3,4,5,6,7}(1,\omega,\omega^2,\ldots,\omega^7) =\\
&4(7,-1-\omega-\omega^2-\omega^3,-1-\omega^2,-1+\omega^2-\omega^3+\omega^5,\\
&-1,-1+\omega-\omega^2+\omega^3,-1+\omega^2,-1+\omega^2+\omega^3-\omega^5).
\end{aligned}$$

Inverzna Furijeova transformacija

Brza Furijeova transformacija rešava samo pola problema: vrednosti zadatih polinoma $P(x)$ i $Q(x)$ mogu se efikasno izračunati u tačkama $1,\omega,\ldots,\omega^{n-1}$, izmnožiti parovi dobijenih vrednosti, i tako naći vrednost polinoma $PQ(x)$ u navedenim tačkama. Ostaje problem interpolacije, odnosno određivanja koeficijenata polinoma PQ na osnovu vrednosti u tačkama $1,w,\ldots,w^{n-1}$. Na sreću,

ispostavlja se da je problem interpolacije vrlo sličan problemu izračunavanja vrednosti, i da ga rešava praktično isti algoritam.

Vratimo se matričnoj notaciji:

$$\begin{pmatrix} 1 & 1 & 1 & \dots & 1 \\ 1 & \omega & \omega^2 & \dots & \omega^{n-1} \\ 1 & \omega^2 & \omega^{2\cdot 2} & \dots & \omega^{2\cdot(n-1)} \\ \dots & \dots & \dots & \dots & \dots \\ 1 & \omega^{n-1} & \omega^{(n-1)\cdot 2} & \dots & \omega^{(n-1)\cdot(n-1)} \end{pmatrix} \begin{pmatrix} a_0 \\ a_1 \\ \dots \\ a_{n-1} \end{pmatrix} = \begin{pmatrix} P(1) \\ P(\omega) \\ \dots \\ P(\omega^{n-1}) \end{pmatrix} \tag{14}$$

Označimo vektor koeficijenata polinoma P sa $\mathbf{a} = (a_0, a_1, \dots, a_{n-1})^T$, vektor vrednosti polinoma sa $\mathbf{v} = (P(1), P(\omega), \dots, P(\omega^{n-1}))^T$, a Vandermondovu matricu sa $V(\omega)$. Ako su zadati koeficijenti polinoma $\mathbf{a}$, njegove vrednosti $\mathbf{v}$ u n tačaka $1, \omega, \dots, \omega^{n-1}$ dobijaju se prema jednačini (14) izračunavanjem proizvoda:

$$\mathbf{v} = V(\omega)\mathbf{a}.$$

S druge strane, ako su zadate vrednosti polinoma

$$\mathbf{v} = (P(1), P(\omega), \dots, P(\omega^{n-1}))^T = (v_0, v_1, \dots, v_{n-1})^T$$

a potrebno je izračunati njegove koeficijente $\mathbf{a}$, jednakost (14) postaje sistem linearnih jednačina po $\mathbf{a}$. Rešavanje sistema jednačina svodi se na računanje inverzne matrice $V(\omega)^{-1}$ algoritmom složenosti $O(n^3)$. S druge strane, ako se zna matrica $V(\omega)^{-1}$, sistem se lako rešava množenjem jednakosti $\mathbf{v} = V(\omega)\mathbf{a}$ s leve strane matricom $V(\omega)^{-1}$:

$$\mathbf{a} = V(\omega)^{-1}\mathbf{v}.$$

Primetimo da Vandermondova matrica ima veoma pravilnu strukturu koja se može iskoristiti za efikasnije izračunavanje njoj inverzne matrice. Neposredno se proverava da je

$$V(\omega)V(\omega^{-1}) = nE,$$

gde je sa E označena jedinična matrica reda n. Zaista, ako je $r \neq s$, onda je za $r, s = 0, \dots, n-1$ proizvod $(r+1)$-e vrste matrice $V(\omega)$ i $(s+1)$-e kolone matrice $V(\omega^{-1})$ jednak

$$\sum_{k=0}^{n-1} \omega^{rk}\omega^{-sk} = \sum_{k=0}^{n-1} \omega^{(r-s)k} = \frac{1-\omega^{n(r-s)}}{1-\omega^{r-s}} = 0.$$

Ako je pak $r = s$, onda je taj proizvod jednak

$$\sum_{k=0}^{n-1} \omega^{rk}\omega^{-rk} = \sum_{k=0}^{n-1} 1 = n.$$

Time je dokazana sledeća teorema.

Teorema: Inverzna matrica matrice $V(\omega)$ Furijeove transformacije jednaka je

$$V(\omega)^{-1} = \frac{1}{n} V(\omega^{-1}).$$

Rešavanje sistema jednačina $\mathbf{v} = V(\omega)\mathbf{a}$ svodi se, dakle, na izračunavanje proizvoda

$$\mathbf{a} = \frac{1}{n} V(\omega^{-1})\mathbf{v}.$$

Posao se dalje pojednostavljuje zahvaljujući sledećoj teoremi.

Teorema: Ako je ω primitivni n-ti koren iz jedinice, onda je ω^{-1} takođe primitivni n-ti koren iz jedinice.

Prema tome, proizvod $\frac{1}{n} V(\omega^{-1})\mathbf{v}$ može se izračunati primenom brze Furijeove transformacije, zamenjujući ω sa ω^{-1} i zatim deljenjem komponenti dobijenog vektora sa n. Ova transformacija zove se *inverzna Furijeova transformacija* (eng. inverse Fourier transform).

Uzimajući sve u obzir, proizvod polinoma A i B stepena n može se izračunati korišćenjem $O(n \log n)$ operacija (sa kompleksnim brojevima) na sledeći način:

- računamo vrednost polinoma A i B u $2n - 1$ tačaka (stepeni $2n$-tog primitivnog korena iz jedinice w) brzom Furijeovom transformacijom,
- u svakoj od tačaka množimo vrednost polinoma A i B,
- računamo brzu Furijeovu transformaciju dobijenog vektora pri čemu umesto vrednosti w koristimo vrednost w^{-1} i rezultat množimo sa $\frac{1}{2n}$.

Implementacija

Razmotrimo ovde pitanje implementacije ovog algoritma. U jeziku C++ kompleksne brojeve imamo na raspolaganju u obliku tipova `complex<double>` i `complex<float>` (zapis u dvostrukoj i jednostrukoj tačnosti). Direktan način implementacije je sledeći.

```
typedef complex<double> Complex;
typedef vector<Complex> ComplexVector;

// brza Furijeova transformacija vektora a duzine n=2^k
// bool parametar inverzna odredjuje da li je direktna ili inverzna
ComplexVector fft(const ComplexVector& a, bool inverzna) {
  // broj koeficijenata polinoma
  int n = a.size();

  // ako je stepen polinoma 0, vrednost u svakoj tacki jednaka
  // je jedinom koeficijentu
  if (n == 1)
    return ComplexVector(1, a[0]);
```

```
  // izdvajamo koeficijente na parnim i na neparnim pozicijama
  ComplexVector A(n / 2), B(n / 2);
  for (int i = 0; i < n / 2; i++) {
    A[i] = a[2 * i];
    B[i] = a[2 * i + 1];
  }

  // rekurzivno izracunavamo Furijeove transformacije tih polinoma
  ComplexVector fftA = fft(A, inverzna),
                fftB = fft(B, inverzna);

  // objedinjujemo rezultat
  ComplexVector rez(n);
  for (int k = 0; k < n; k++) {
    // odredjujemo primitivni n-ti koren iz jedinice
    double coeff = inverzna ? -1.0 : 1.0;
    complex<double> w = exp((coeff * 2 * k * M_PI / n) * 1i);
    // racunamo vrednost polinoma u toj tacki
    rez[k] = fftA[k % (n / 2)] + w * fftB[k % (n / 2)];
  }
  // vracamo konacan rezultat
  return rez;
}


// funkcija vrsi direktnu Furijeovu transformaciju polinoma ciji su
// koeficijenti odredjeni nizom a duzine 2^k
ComplexVector fft(const ComplexVector& a) {
  return fft(a, false);
}

// funkcija vrsi inverznu Furijeovu transformaciju polinoma ciji su
// koeficijenti odredjeni nizom a duzine 2^k
ComplexVector ifft(const ComplexVector& a) {
  ComplexVector rez = fft(a, true);
  // nakon izracunavanja vrednosti, potrebno je jos podeliti
  // sve koeficijente duzinom vektora
  int n = a.size();
  for (int k = 0; k < n; k++)
    rez[k] /= n;
  return rez;
}
```

Jasno je da algoritam realizovan prethodnom procedurom zadovoljava jednačinu $T(n) = 2T(n/2) + O(n)$, pa joj je složenost $O(n \log n)$. Nakon transformacije (promene reprezentacije izračunavanjem vrednosti) dva polinoma njihovo

množenje je moguće u vremenu $O(n)$, pa je ukupna složenost množenja polinoma pomoću FFT jednaka $O(n \log n)$.

Međutim, prethodnu direktnu implementaciju je moguće poboljšati. Najveći problem je to što se primitivni n-ti koreni iz jedinice računaju zasebno u svakom rekurzivnom pozivu. Efikasnost bi se značajno popravila ako bi se niz korena izračunao samo jednom, pre funkcije. Problem je u tome što se tokom rekurzije n smanjuje, pa su nam u svakom pozivu potrebni različiti koreni. Međutim, ako je neki broj k-ti primitivni koren iz jedinice, onda je on i $2k$-ti primitivni koren iz jedinice. Zato, ako znamo niz primitivnih n-tih korena iz jedinice ($e^{\frac{2k\pi i}{n}}$, za k od 0 do $n-1$), tada su elementi na parnim pozicijama tog vektora $n/2$-ti primitivni koreni iz jedinice. Zaista, ako je $k = 2k'$, tada je $e^{\frac{2k\pi i}{n}} = e^{\frac{2k'\pi i}{\frac{n}{2}}}$ i važi da ako je $0 \leq k < n$, tada je $0 \leq k' < n/2$. Dakle, u početku možemo izračunati niz svih primitivnih n-tih korena iz jedinice i njih koristiti na početnom nivou rekurzije, na narednom nivou rekurzije ćemo koristiti svaki drugi element tog vektora, na narednom svaki četvrti, zatim svaki osmi i tako dalje. Na primer, ako je $n = 8$, početni niz korena je 1, $\frac{\sqrt{2}}{2} + i\frac{\sqrt{2}}{2}$, i, $-\frac{\sqrt{2}}{2} + i\frac{\sqrt{2}}{2}$, -1, $-\frac{\sqrt{2}}{2} - i\frac{\sqrt{2}}{2}$, $-i$, $\frac{\sqrt{2}}{2} - i\frac{\sqrt{2}}{2}$, dok je na narednom nivou rekurzije $n = 4$ i koriste se koreni 1, i, -1, $-i$, dok je na narednom nivou rekurzije $i = 2$ i koriste se 1 i -1.

Još jedan problem prethodne implementacije je to što se tokom rekurzije alociraju i popunjavaju pomoćni vektori, što dovodi do gubitka i vremena i memorije. Furijeovu transformaciju je moguće realizovati i bez korišćenja pomoćne memorije. Na početnom nivou rekurzije koeficijenti ulaznog polinoma su svi dati početnim vektorima koeficijenata. Na narednom se posmatraju elementi na pozicijama 0, 2, 4, ..., $n-2$ i na pozicijama 1, 3, ..., $n-1$. Na narednom se posmatraju elementi na pozicijama 0, 4, 8, $n-4$, zatim elementi na pozicijama 1, 5, ..., $n-3$, zatim elemeni na pozicijama 2, 6, ..., $n-2$, i na kraju elementi na pozicijama 3, 7, ..., $n-1$. Slično se nastavlja i na daljim nivoima rekurzije. Dakle, umesto formiranja pomoćnog ulaznog vektora sa pogodno odabranim ulaznim koeficijentima, prosleđivaćemo originalni vektor, poziciju početka s i pomeraj d i posmatraćemo njegove elemente na pozicijama $s + dk$, za $0 \leq k < n$, gde je $n = n_0/d$, a n_0 je dužina početnog vektora. Rezultate rekurzivnih poziva možemo smestiti u dve polovine rezultujućeg niza. Nakon toga rezultate objedinjavamo. Vrednosti na pozicijama k i $k+n/2$ rezultujućeg vektora određene su vrednostima na poziciji k u rezultatu prvog i drugog rekurzivnog, međutim, one se nalaze upravo na pozicijama k i $k + n/2$ rezultujućeg vektora (jer smo rezultate rekurzivnih poziva smestili u prvu i drugu polovinu rezultata). Moramo voditi računa da te dve vrednosti moramo istovremeno izračunati i ažurirati.

```
typedef complex<double> Complex;
typedef vector<Complex> ComplexVector;

// funkcija vrši Furijeovu transformaciju (direkntu ili inverznu) elemenata
//   a[start_a], a[start_a + step], a[start_a + 2step], ...
// i rezultat smešta u niz
```

```
//   rez[start_rez], rez[start_rez + 1], rez[start_rez + 2], ...
// koristeci primitivne korene iz jedinice smestene u niz
//   w[0], w[step], w[2step], ...
void fft(const ComplexVector& a, int start_a,
         const ComplexVector& w,
         ComplexVector& rez, int start_rez,
         int step) {
  // broj elemenata niza koji se transformise
  int n = a.size() / step;

  // stepen polinoma je nula, pa mu je vrednost u svakoj tacki jednaka
  // konstantnom koeficijentu
  if (n == 1) {
    rez[start_rez] = a[start_a];
    return;
  }

  // rekurzivno transformisemo niz koeficijenata na parnim pozicijama
  // smestajuci rezultat u prvu polovinu niza rez
  fft(a, start_a, w, rez, start_rez, step*2);
  // rekurzivno transformisemo niz koeficijenata na neparnim pozicijama
  // smestajuci rezultat u drugu polovinu niza rez
  fft(a, start_a + step, w, rez, start_rez + n/2, step*2);

  // objedinjujemo dve polovine u rezultujuci niz
  for (int i = 0; i < n/2; i++) {
    auto r1 = rez[start_rez + i];
    auto r2 = rez[start_rez + (i + n/2)];
    rez[start_rez + i] = r1 + w[i*step] * r2;
    rez[start_rez + (i + n/2)] = r1 - w[i*step] * r2;
  }
}

// funkcija vrsi direktnu Furijeovu transformaciju polinoma ciji su
// koeficijenti odredjeni nizom a duzine 2^k
ComplexVector fft(const ComplexVector& a) {
  // duzina niza koeficijenata polinoma
  int n = a.size();
  // izracunavamo primitivne n-te korene iz jedinice
  ComplexVector w(n/2);
  for (int k = 0; k < n/2; k++)
    w[k] = exp((2 * k * M_PI / n) * 1i);
  // vektor u koji se smesta rezultat
  ComplexVector rez(n);
  // vrsimo transformaciju
  fft(a, 0, w, rez, 0, 1);
```

```
  // vracamo dobijeni rezultat
  return rez;
}

// funkcija vrsi inverznu Furijeovu transformaciju polinoma ciji su
// koeficijenti odredjeni nizom a duzine 2^k
ComplexVector ifft(const ComplexVector& a) {
  // duzina niza koeficijenata polinoma
  int n = a.size();
  // izracunavamo primitivne n-te korene iz jedinice
  ComplexVector w(n);
  for (int k = 0; k < n; k++)
    w[k] = exp((- 2 * k * M_PI / n) * 1i);
  // vektor u koji se smesta rezultat
  ComplexVector rez(n);
  // vrsimo transformaciju
  fft(a, 0, w, rez, 0, 1);
  // popravljamo rezultat
  for (int i = 0; i < n; i++)
    rez[i] /= n;
  // vracamo dobijeni rezultat
  return rez;
}

vector<double> proizvod(const vector<double>& p1,
                        const vector<double>& p2) {
  // duzina niza koeficijenata
  int n = p1.size();
  // kreiramo nizove kompleksnih koeficijenata dopunjavajuci ih nulama
  // do dvostruke duzine
  int N = 2*n;
  ComplexVector a(N, 0.0);
  copy(begin(p1), end(p1), begin(a));
  ComplexVector b(N, 0.0);
  copy(begin(p2), end(p2), begin(b));

  // vrsimo Furijeove transformacije oba vektora koeficijenata
  ComplexVector va = fft(a), vb = fft(b);
  // prolazimo Furijeovu transformaciju vektora koeficijenata proizvoda
  ComplexVector vc(N);
  for (int i = 0; i < N; i++)
    vc[i] = va[i] * vb[i];
  // inverznom Furijeovom transformacijom rekonstruisemo koeficijente
  // proizvoda
  ComplexVector c = ifft(vc);
```

```
  // realne delove kompleksnih brojeva smestamo u niz rez i vracamo ga
  vector<double> rez(N);
  transform(begin(c), end(c), begin(rez),
            [](Complex x){
              return real(x);
            });
  return rez;
}
```

Pažljivom analizom je moguće ukloniti rekurziju iz prethodne implementacije (tako se dobija Kuli-Tukijev nerekurzivni algoritam za FFT, zasnovan na tzv. leptir-shemi).