UNIVERZITET U BEOGRADU
MATEMATIČKI FAKULTET

PARALELIZACIJA ALGORITAMA ZA MNOŽENJE VELIKIH BROJEVA I NJIHOVA PRIMENA NA ISPITIVANJE PRIMALNOSTI FERMAOVIH BROJEVA

-MAGISTARSKI RAD-

MILENKO R. MOSUROVIĆ

Beograd, 1996. godine

Komisija:
1. Ž. Mijajlović (mentor)
2. S. Prešić
3. A. Ivić
4. M. Živković

Pitanja: 1. Izvođenje algoritma na procesorima većim od 2^{16} proc.
2. Drugi problemi teorije brojeva koji se mogu rešavati paralelnim računarima
3. $k \cdot 2^n + 1$
4. Koji model paralelnih mašina; Šenhage - Štrasen

Mentor:

Dr Žarko Mijajlović
Univerzitet u Beogradu
Matematički fakultet
Beograd

Članovi komisije:

Dr Slaviša Prešić
Univerzitet u Beogradu
Matematički fakultet
Beograd

Dr Aleksandar Ivić
Univerzitet u Beogradu
Rudarsko Geološki fakultet
Beograd

Dr Miodrag Živković
Univerzitet u Beogradu
Matematički fakultet
Beograd

Datum odbrane:

SADRŽAJ

strana

Predgovor

Ferma je postavio hipotezu da su brojevi oblika $F_n = 2^{2^n} + 1$, gdje je n proizvoljan ceo nenegativan broj, prosti. Takvi brojevi su po njemu dobili ime Fermaovi brojevi. Hipoteza je tačna za n od 0 do 4. Međutim, Ojler je uočio da 641 deli F_5 i time dokazao da je Fermaova hipoteza pogrešna. Otuda je postalo zanimljivo, da se za dato n utvrdi karakter broja F_n. Za sada, prema nama dostupnim podacima, postoje samo parcijalna rešenja, tj. dokazano je da su neki od brojeva F_n ($n \geq 5$) složeni (vidi [5], [10]). Za F_{14}, F_{20} i F_{22} zna se samo da su složeni, dok im se faktori ne znaju. Za ispitivanje karaktera ovih brojeva korišćen je Pepinov test: za $n \geq 1$, F_n je prost ako i samo ako je $3^{(F_n-1)/2} \equiv -1 (\mathrm{mod}\ F_n)$. Prvi Fermaov broj nepoznatog karaktera, prema nama dostupnim podacima, je F_{24}, a potom F_{28} (vidi [5], [10]).

U radu [5] se kaže da je za ispitivanje karaktera broja F_{22} utrošeno nešto više od sedam meseci, dok bi za ispitivanje karaktera broja F_{24} trebalo oko deset godina (misli se na algoritam i opremu koja je korišćena za ispitivanje karaktera broja F_{22}). Ova procena se zasniva na činjenici da ako se n poveća za jedan onda se vreme potrebno za Pepinov test poveća oko 4 puta. Otuda, možemo zaključiti da bi nam za ispitivanje karaktera broja F_{28} trebalo više od dvije i po hiljade godina. Napomenimo da je stvar još lošija, jer se algoritam korišćen za testiranje broja F_{22} ne može upotrebiti za testiranje broja F_{28} bez suštinskih promena (mašine ili algoritma).

Ovaj rad je proistekao iz ideje da se predlože paralelni algoritmi i model (realan) na kom bi se oni izvršavali, kako bi smanjili vreme potrebno za testiranje karaktera prethodno pomenutih brojeva. Osnovna operacija Pepinovog testa je kvadriranje velikih brojeva, i ta operacija se ponavlja mnogo puta. Otuda, problem paralelizacije Pepinovog testa, svodi se na problem paralelizacije algoritama za množenje (kvadriranje) velikih brojeva, što je središna tema ovoga rada. Pri tome smo se opredelili za paralelizaciju Šenhage-Štrasenovog algoritma (vidi [1]) koji je asimptotski najbolji poznati sekvencijalni algoritam.

Rad je podeljen na tri glave:

1. Uvod,
2. Algoritmi za množenje velikih brojeva i
3. Paralelizacija algoritama.

U uvodu se navode pojmovi i tvrđenja, koje koristimo pri našem daljem izlaganju. Naime, opisujemo arhitekture paralelnih računara, posebno njihovu podelu zasnovanu na konceptima toka instrukcija i toka podataka, kao i analizu algoritama, stavljajući naglasak na analizi paralelnih algoritama. Zatim navodimo teoreme i pojmove iz teorije brojeva koji su nam neophodni da bi izveli dokaz Pepinove teoreme. Na kraju uvodne glave izlažemo diskretnu Furijeovu transformaciju. Pri tome, navodimo dokaz konvolucione teoreme, kao i niz tvrđenja koja omogućavaju računanje Furijeove transformacije u

prstenu celih brojeva po modulu F_n. Prikazan je i FFT algoritam.

U drugoj glavi dajemo kratak pregled nekih sekvencijalnih algoritama i njihove vremenske složenosti za množenje velikih brojeva, kao što su: klasični (školski) algoritam, Karacuba-Hofmanov algoritam, Šenhage-Štrasenov algoritam, i algoritam zasnovan na diskretnoj težinskoj transformaciji. Svakako za nas je najvažniji Šenhage-Štrasenov algoritam koji ima najmanje asimptotsko vreme izvršavanja.

U okviru treće glave, koja delom sadrži orginalni tekst, izlažemo paralelizaciju Šenhage-Štrasenovog algoritma. Rezultat do kojeg smo došli je da množenje (Pepinov test) možemo ubrzati linearno po broju procesora. Ovako dobar rezultat je postignut uravnoteženim punjenjem procesora podacima, na taj način procesori sadrže (obrađuju) istu količinu podataka, kao i izborom pogodnog modela, tako da se komunikacije obavljaju uglavnom između susednih procesora. Sem toga, u poglavlju (3.2), daje se predlog kako u nekim slučajevima, može biti pojednostavljen Šenhage-Štrasenov algoritam, što je na primer korisno u testiranju karaktera brojeva oblika $h2^n + 1$[1], kao i samoj paralelizaciji Šenhage-Štrasenovog algoritma. Na kraju ove glave daje se kratak pregled implementacije predloženih algoritama na paralelnom računaru baziranom na T800 procesorima, koje prikazujemo u prilogu.

*****

Najveću zahvalnost dugujem svom mentoru prof. dr Žarku Mijajloviću, koji mi je pomogao da izaberem jednu zaista interesantnu temu, a u toku izrade teze, svojim savetima uticao da prevaziđem, manje ili veće, poteškoće u razjašnjenju pojedinih pitanja. Njegova izuzetna angažovanost bila mi je dragocena i davala mi veliki podsticaj u radu.

Zahvaljujem se dr. Miodragu Živkoviću koji je, svojim korisnim primedbama, omogućio da se poboljša konačna verzija teksta.

Koristim priliku da se zahvalim i Matematičkom institutu u Beogradu, na uslovima i pomoći koja mi je bila obezbeđena pri korišćenju paralelnog računara baziranog na T800 procesorima (ovaj računar je njihovo vlasništvo). Posebno se zahvaljujem Mr. Draganu Uroševiću.

Takođe želim da se zahvalim i svom matičnom fakultetu, PMF-u u Podgorici za pomoć pruženu pri izradi ovog rada.

Blagodarim i svima koji su mi na razne načine pomogli da započnem i okončam ovaj rad.

Beograd 1996. Milenko R. Mosurović

[1] Vidi Teoremu 1.8.

1 Uvod

1.1 Modeli izračunavanja i analiza algoritama

Arhikteture paralelnih računara

Postoji više načina za klasifikaciju arhitektura paralelnih računara. Detaljni prikaz raznih arhiktektura i načina klasifikacije može se naći u radu [7].

Ovde ćemo izložiti podelu, zasnovanu na konceptima toka instrukcija (instruction stream) i toka podataka (data stream), opisanu u [2]. Tok instrukcija je niz instrukcija koje računar treba da izvrši, a tok podataka je niz podataka nad kojim ove instrukcije operišu.

Zavisno od toga da li postoji jedan ili više tokova instrukcija ili podataka podela je izvršena na SISD, MISD, SIMD i MIMD računare.

SISD(Single Instruction stream, Single Data stream) računari imaju jedan procesor koji izvršava jedan tok instrukcija koji obrađuje jedan tok podataka. Većina serijskih računara je iz ove grupe.

MISD (Multiple Instruction stream, Single Data stream) računari imaju n procesora , $n > 1$. Svaki procesor ima svoju kontrolnu jedinicu. Podaci su smešteni u zajedničkoj memoriji. Na svakom koraku procesor dobije podatak iz memorije i obrađuje ga instrukcijom koju je poslala njegova kontrolna jedinica. Dakle, nad jednim tokom podataka simultano se izvršava više tokova instrukcija.

SIMD (Single Instruction stream, Multiple Data stream) računari imaju n procesora , $n > 1$. Svaki procesor ima svoju memoriju. U tim lokalnim memorijama mogu biti smešteni programi i podaci. Rad procesora se odvija prema jednom toku instrukcija koje odašilje jedna kontrolna jedinica. U jednom koraku svi procesori dobijaju istu instrukciju koju izvršavaju nad nekim podatkom iz memorije. Ne moraju svi procesori da izvršavaju sve instrukcije koje dobiju. Ako neki procesor treba da preskoči neku instrukciju (to obaveštenje nosi sama instrukcija) ona čeka ostale da je završe. Čekanje postoji i kada neki od procesora završi neku instrukciju pre nego ostali. Na taj način se postiže sinhronizovani rad procesora.

MIMD (Multiple Instruction stream, Multiple Data stream) računari imaju n, $n > 1$ procesora. Svaki procesor izvršava svoj tok instrukcija nad svojim tokom podataka. Dakle, rad ovih procesora nije sinhronizovan.

U većini slučajeva prilikom izvršavanja nekog algoritma na SIMD ili MIMD mašini javlja se potreba da procesori međusobno komuniciraju tj. da šalju jedni drugima podatke.

Načini komuniciranja među procesorima SIMD (MIMD) mašine daju podelu na SM (Shared Memory) SIMD (MIMD) i mrežne modele (Interconnection Network) SIMD (MIMD) računare.

Kod SM SIMD (MIMD) računara procesori komuniciraju preko zajedničke memorije. Ako su memorijske lokacije kojima procesori prilaze, zbog upisivanja ili čitanja podataka, različite onda se dozvoljava simultani pristup.

Zavisno od toga kako procesori mogu prići istoj memorijskoj lokaciji SM SIMD (MIMD) mašine se dele na:

EREW (Exclusive-Read, Exclusive-Write) SM SIMD (MIMD) računari. Nema simultanog pristupa od različitih procesora istoj memorijskoj lokaciji ni zbog čitanja ni zbog upisivanja.

ERCW (Exclusive-Read, Concurrent-Write) SM SIMD (MIMD) računari. Može se simultano upisivati ali ne i čitati.

CREW (Concurrent-Read, Exclusive-Write) SM SIMD (MIMD) računari. Može se simultano čitati ali ne i upisivati.

CRCW (Concurrent-Read, Concurrent-Write) SM SIMD (MIMD) računari. Dozvoljeno je simultano upisivanje i čitanje.

SM MIMD modeli su poznati pod imenom multiprocesori (tightily coupled machines).

Mrežni SIMD (MIMD) model omogućava razmenu podataka među procesorima preko komunikacionih linija kojima su procesori povezani.

Kod potpune mreže svaki procesor ima vezu sa svim ostalim. Obično, ovolika povezanost nije potrebna pa su popularniji slabije povezani modeli: linearni niz procesora, dvodimenzionalni niz procesora, drvo, kub itd.

Kod linearnog procesorskog niza procesori $P_1, ..., P_n$ su povezani dvosmernim komunikacionim linijama tako da procesor P_1 ima vezu sa P_2, procesor P_n vezu sa P_{n-1} i procesor P_k, $1 < k < n$, vezu sa P_{k-1} i P_{k+1}.

Dvodimenzionalni procesorski niz (mesh) se sastoji od n procesora razmeštenih u obliku $k \times k$ matrice, gdje je $k = \sqrt{n}$. Procesor $P_{i,j}$ se nalazi u $i-$toj vrsti i $j-$toj koloni matrice. Unutrašnji procesori imaju vezu sa susedima $P_{i+1,j}, P_{i-1,j}, P_{i,j+1}$ i $P_{i,j-1}$. Procesori na granici imaju manje veza, ugaoni po dve a ostali po tri veze.

Pretpostavimo da je $n = 2^s, s \leq 1$ tada mrežu od n procesora u kojoj je svaki procesor povezan sa s susednih nazivamo $s-$dimenzionalni kub ili $s-$dimenzionalna hiperkocka.

Mrežni MIMD računari su poznati pod imenom multikompjuteri (loosely coupled machines).

Analiza algoritama

Ako imamo više algoritama koji rešavaju jedan problem, mi treba da donesemo odluku koji od njih da izaberemo. Da bi odluka bila ispravna mi moramo imati neke kriterijume, koji će nam ukazati da je jedan algoritam "bolji" od drugog. Kriterijumi mogu biti različiti, ali se vreme izvršavanja (vremenska složenost), kao i prostor (memorijski) koji algoritam zahteva (prostorna složenost) najčešće koriste.

Cilj analize algoritma je predviđanje ponašanja algoritma, specijalno vremena izvršavanja, bez njegove implementacije na specijalnom računaru. Ponašanje algoritma zavisi od nekih parametara, najčešće od veličine ulaza, pa se i izražava kao funkcija veličine ulaza. Obično nas interesuje ponašanje

algoritma kad veličina ulaza raste.

Da bi smo predvideli vreme izvršavanja algoritma, mi možemo prebrojati osnovne operacije, *korake* koje algoritam treba da izvrši. Pri tome glavni metod je upotreba aproksimacija i obično se koristi $O(\cdot)$ notacija. Ignorišu se mnogi detalji a ističu se samo bitne karakteristike algoritma. Tj. ne moramo brojati sve korake već samo glavne, naprimer broj operacija poređenja kod algoritama za sortiranje baziranih na poređenju.

Precizne definicije pojmova vezane za složenost sekvencijalnih algoritama kao i $O(\cdot)$ notacije, mogu se naći u [1] [9]. Pri našim daljim izlaganjima mi ćemo pod pojmom korak imati u vidu instrukcije koje se mogu izvršiti na modelu koji je blizak savremenom računaru.

Koristićemo oznaku $O_B(\cdot)$ da bi ukazali da smo kao meru složenosti uzeli broj operacija koje se izvode na bitovima tj. broj bit operacija. (Za preciznu definiciju vidi [1].)

U mnogim slučajevima, posebno kada se analiziraju rekurzivni algoritmi, javljaju se rekurentne relacije. Razvijene su posebne tehnike za rešavanje pojedinih rekurentnih relacija. Teorema koju navodimo niže, omogućava nam da rešavamo rekurentne relacije koje će se pojaviti pri analizi algoritama koje kasnije izlažemo.

Teorema 1.1. *Rešenje rekurentne relacije* $T(n) = aT(n/b) + cn^k$, *gdje su* a *i* b *celobrojne konstante,* $a \geq 1$, $b \geq 2$, a c *i* k *su pozitivne konstante, je*

$$T(n) = \begin{cases} O(n^{\log_b a}) & \text{ako je } a > b^k \\ O(n^k \log n) & \text{ako je } a = b^k \\ O(n^k) & \text{ako je } a < b^k \end{cases}$$

Rekurentna relacija iz Teoreme (1.1.) se pojavljuje pri analizi mnogih algoritama baziranih na strategiji *podeli pa vladaj*. Dokaz Teoreme (1.1.) se može naći u [9].

Pri ocenjivanju paralelnih algoritama kriterijumi koji se najčešće koriste su: vreme izvršavanja, broj procesora koji se upotrebljava i cena (vidi [2]).

Kako je jedan od glavnih razloga za uvođenje paralelizma povećanje brzine izvršavanja, to je nesumnjivo najznačajnija mera pri ocenjivanju paralelnih algoritama njihovo *vreme izvršavanja*. Ono se definiše kao vreme koje zahteva algoritam da bi se rešio problem na paralelnom računaru, tj. vreme koje protekne od momenta kada je algoritam (prvi procesor) počeo sa radom do momenta kada se algoritam (i poslednji procesor) zaustavi.

Vreme izvršavanja paralelnog algoritma ćemo označavati sa $T(n, p)$, gdje je n veličina ulaza, a p broj procesora.

Predviđanje vremena izvršavanja opet možemo ostvariti brojanjem koraka koje algoritam treba da izvrši. Pri tome treba razlikovati dve vrste koraka: *računski koraci* i *koraci usmeravanja*. *Računski korak* je aritmetička ili logička operacija koja se izvodi nad podacima unutar procesora. Dok u *koraku usmeravanja* podatak putuje od jednog procesora do drugog kroz

zajedničku memoriju ili komunikacionu mrežu. Obično koraci usmeravanja zahtevaju nešto više vremena za izvršavanje nego računski koraci, ali radi jednostavnije analize mi ćemo pretpostaviti da oni zahtevaju isto vreme.

Pri ocenjivanju paralelnih algoritama za dati problem, sasvim je prirodno da se to čini u odnosu na najbolji poznati sekvencijalni algoritam za taj problem. Tako dobra indikacija kvaliteta paralelnog algoritma je *ubrzanje (speedup)* koje se postiže. Za dati problem ovo je definisano kao

$$S(p) = \frac{T(n,1)}{T(n,p)}.$$

Jasno je da za veće ubrzanje imamo bolji algoritam. Treba napomenuti da ubrzanje ne može biti veće od broja procesora, jer bi u suprotnom postojao brži sekvencijalni algoritam.

Cena (cost) paralelnog algoritma je definisana kao:

$$C(n,p) = T(n,p) \times p.$$

Ili drugim rečima, cena je jednaka broju koraka izvršenih ukupnō na svim procesorima pri rešavanju datog problema. Ovo u slučaju da svi procesori izvršavaju isti broj koraka. Ako to nije slučaj onda cena predstavlja gornju granicu ukupnog broja izvršenih koraka i tada se uvodi dodatna mera *produktivnost (efficiency)* . Produktivnost (ili efikasnost) se definiše kao:

$$E(n,p) = \frac{T(n,1)}{pT(n,p)}.$$

Produktivnost, iskazuje stepen uposlenosti pojedinih procesora pri rešavanju datog problema i uvek je ≤ 1. Napomenimo da se u nekim slučajevima efikasnost može povećati smanjenjem broja procesora i većim upošljavanjem preostalih procesora, a da se vreme izvršavanja promeni najviše za konstantan faktor (Brentova lema, vidi [9]).

Svakako postoje i drugi kriterijumi za ocenjivanje paralelnih algoritama kojima se mi nećemo baviti.

1.2 Ispitivanje primalnosti i Fermaovi brojevi

U ovoj glavi navedene su teoreme (vidi [10]) koje se koriste za dokazivanje primalnosti. Pre formulacije ovih teorema navodimo i neke pojmove i pomoćne teoreme iz teorije brojeva (vidi [10]) koje koristimo.

Definicija 1.1. *Skup brojeva M se zove modul ako važi: Ako su $x, y \in M$ tada su $(x-y)$, $(x+y) \in M$.*

Teorema 1.2. *Svi elementi modula M koji sadrži samo cele brojeve, su višestrukosti određenog broja d (tj. mogu se napisati kao $k \cdot d$, gdje je k ceo broj), koji je najmanji pozitivan ceo broj iz M. Izuzetak je modul koji sadrži samo nulu.*

Teorema 1.3 (Ojlerova Teorema) *Ako je* $(a,n) = 1$ *i* $n = \prod p_i^{\alpha_i}$, *tada je* $a^{\varphi(n)} \equiv 1(\text{mod } n)$, *gdje je* $\varphi(n) = \prod p_i^{\alpha_i - 1}(p_i - 1)$ *broj relativno prostih brojeva sa* n, *a manjih od* n.

Neposredna posledica Teoreme (1.3) je računanje inverznog elementa tj.

$$b^{-1} \equiv b^{\varphi(n)-1}(\text{mod } n).$$

Definicija 1.2. *Ako je* $(a,n) = 1$ *i ako kongruencija*

$$x^2 \equiv a(\text{mod } n) \tag{1.2.1}$$

ima rešenje x, *tada se* a *zove kvadratni ostatak broja* n. *Ako kongruencija (1.2.1) nema rešenje onda za* a *kažemo da je kvadratni ne-ostatak broja* n.

Za slučaj kada je n neparan prost broj p, Legendre je uveo poseban simbol

$$(\frac{a}{p}) = \begin{cases} 1 & \text{ako je } a \text{ kvadratni ostatak} \\ -1 & \text{ako je } a \text{ kvadratni ne-ostatak} \end{cases}$$

Teorema 1.4 (Ojlerov kriterijum) *Ako je* $(a,p) = 1$ *i* p *je neparan prost broj, tada je*

$$(\frac{a}{p}) = a^{(p-1)/2}(\text{mod } p).$$

Teorema 1.5. *Ako su* p *i* q *neparni prosti brojevi tada je*

$$(\frac{p}{q}) = (\frac{q}{p})(-1)^{\frac{1}{2}(p-1)\frac{1}{2}(q-1)}.$$

Jedan način dokazivanja primalnosti je korišćenjem Lehmerove teoreme (vidi [10])

Teorema 1.6. *Pretpostavimo da je* $N - 1 = \prod_{j=1}^{n} q_j^{\beta_j}$, *gdje su* q_j *različiti prosti brojevi. Ako možemo naći ceo broj* a, *takav da je*

$$a^{(N-1)/q_j} \not\equiv 1(\text{mod } N) \quad \textit{za sve } j = 1, 2..., n \tag{1.2.2}$$

i takav da je

$$a^{N-1} \equiv 1(\text{mod } N), \tag{1.2.3}$$

tada je N *prost.*

Dokaz. Posmatrajmo sve stepene e takve da je $a^e \equiv 1(\text{mod } N)$. Ovi stepeni čine modul M koji se sastoji samo od celih brojeva. Zbog (1.2.3) $N - 1$ je element modula M. Po Teoremi (1.2.) modul M je generisan sa celim brojem $d \leq N - 1$ koji deli $N - 1$. Ali svaki delilac d broja $N - 1$, s izuzetkom samog broja $N - 1$, je delilac najmanje jednog od brojeva $(N - 1)/q_j$, $j = 1, 2, ..., n$. Ako bi $d < N - 1$, tada bi najmanje jedan od brojeva $(N - 1)/q_j$ pripadao

modulu M, tj. $a^{(N-1)/q_j} \equiv 1(\text{mod } N)$, što je u suprotnosti sa (1.2.2). Dakle, generator modula M je $N-1$.

S druge strane, Teorema (1.3.) nam kaže da je uvek $a^{\varphi(N)} \equiv 1(\text{mod } N)$, ako je $(a,N)=1$ i da je $\varphi(N) < N-1$ za sve složene brojeve N. Ali ova činjenica nam kaže da $\varphi(N)$ pripada modulu M, što je nemoguće ako je $\varphi(N) < N-1$, jer je generator d najmanji pozitivan broj u modulu. Dakle N mora biti prost.

Napomenimo da ako bi $(a,N) > 1$ tada bi $a^{N-1} \not\equiv 1(\text{mod } N)$, zato je $(a,N)=1$ □

Pepinova teorema za Fermaove brojeve

Lehmerova teorema je sasvim jednostavna kada $N-1$ ima mali broj različitih prostih faktora. Najprostiji slučaj je kada je $N-1$ stepen 2, tj. kada je $N = 2^s+1$. Međutim jednostavno se dokazuje da su ovi brojevi složeni, osim u slučaju kada je s stepen dvojke. Brojevi

$$F_n = 2^{2^n}+1 \tag{1.2.4}$$

se zovu Fermaovi brojevi, i oni mogu biti prosti. Proučimo sada kakvi su zahtevi Lehmerove teoreme da bi broj F_n bio prost. Moramo pronaći broj a takav da je

$$\begin{cases} a^{(F_n-1)/2} = a^{2^{2^n}-1} \not\equiv 1(\text{mod } F_n) \text{ i} \\ a^{F_n-1} = a^{2^{2^n}} \equiv 1(\text{mod } F_n) \end{cases} \tag{1.2.5}$$

Ako stavimo $a^{2^{2^n}-1} = x$, tada imamo

$$x^2 \equiv 1(\text{mod } F_n) \text{ i } x \not\equiv 1(\text{mod } F_n). \tag{1.2.6}$$

Sada, ako je F_n prost, tada je x element prstena bez delitelja nule. Prema tome kongruencija $x^2-1 \equiv (x+1)(x-1) \equiv 0(\text{mod } F_n)$ će imati dva i samo dva rešenja $x \equiv 1(\text{mod } F_n)$ i $x \equiv -1(\text{mod } F_n)$. Pošto rešenje $x \equiv 1(\text{mod } F_n)$ je u suprotnosti sa prvim uslovom iz (1.2.5.), jedina preostala mogućnost je $x \equiv -1(\text{mod } F_n)$. Tako mi tražimo ceo broj a koji zadovoljava

$$x \equiv a^{(F_n-1)/2} \equiv -1(\text{mod } F_n) \tag{1.2.7}$$

kada je F_n prost. Sada Ojlerov kriterijum (Teorema 1.4.), iz teorije kvadratnog ostatka, nam kaže da a mora biti kvadratni ne-ostatak broja F_n. Međutim, uz pomoć Teoreme (1.5.) lako je pokazati da je broj 3 kvadratni ne-ostatak svih prostih brojeva oblika $12n \pm 5$. F_n je baš ovakvog oblika. Zaista: $2^{2^1} = 4$, $2^{2^2} = 16 \equiv 4(\text{mod } 12), \ldots$, tako da $F_n \equiv 4+1 \equiv 5(\text{mod } 12)$. Prema tome ako je F_n prost tada je sigurno $a=3$ kvadratni ne-ostatak broja F_n i time smo dokazali sledeću teoremu (vidi [10]):

Teorema 1.7 (Pepinova Teorema) *Potreban i dovoljan uslov da Fermaov broj $F_n = 2^{2^n} + 1, n \geq 1$, bude prost je da*

$$3^{2^{2^n-1}} \equiv -1 (\text{mod } F_n). \tag{1.2.8}$$

Dakle, da bi ispitali je li F_n prost potrebno je i dovoljno da počev od 3 uzastopno vršimo kvadriranje $2^n - 1$ put, i to po modulu F_n. Broj F_n se povećava ogromnom brzinom za malo povećanje broja n. Zbog toga imamo posla sa velikim brojevima tj. potrebni su nam algoritmi za kvadriranje (množenje) velikih brojeva.

Navedimo još i Protovu teoremu (vidi [10]) koja je analogna Pepinovoj.

Teorema 1.8. *Pretpostavimo da N ima oblik $N = h2^n + 1$, gdje je $2^n > h$ i h neparan ceo broj. Ako postoji ceo broj a takav da je*

$$a^{(N-1)/2} \equiv -1 (\text{mod } N), \tag{1.2.9}$$

tada je N prost.

1.3 Diskretna Furijeova transformacija

Veći deo teksta u ovom poglavlju je preuzet iz knjige [1]. Diskretna Furijeova transformacija (DFT) se obično definiše nad kompleksnim brojevima. Mi ćemo, zbog potrebe u Šenhage-Štrasenovom algoritmu, definisati DFT nad proizvoljnom komutativnom prstenu $(R, +, *, 0, 1)$. Za element ω iz R koji zadovoljava

1. $\omega \neq 1$,
2. $\omega^n = 1$, i
3. $\sum_{j=0}^{n-1} \omega^{jp} = 0$, za $1 \leq p < n$,

kažemo da je *primitivni $n-$ti koren iz jedinice*. Elementi $\omega^0, \omega^1, ..., \omega^{n-1}$ su *$n-$ti koreni iz jedinice*.

Na primer, $e^{2\pi i/n}$, gdje je $i = \sqrt{-1}$, je primitivni $n-$ti koren iz jedinice u prstenu kompleksnih brojeva.

Neka je $a = [a_0, a_1, ..., a_n]^T$ vektor kolona dužine n čiji su elementi iz prstena R. Mi pretpostavljamo da broj n ima multiplikativni inverzni element u R i da R ima primitivni $n-$ti koren iz jedinice ω. Neka A bude $n \times n$ matrica čiji su elementi $A[i, j] = \omega^{ij}$, za $0 \leq i, j < n$. Vektor $F(a) = Aa$ čija je $i-$ta komponenta $b_i = \sum_{k=0}^{n-1} a_k \omega^{ij}$, $0 \leq i < n$ se zove diskretna Furijeova transformacija vektora a. Matrica A je regularna, tj. postoji A^{-1}. A^{-1} je jednostavnog oblika što daje sledeća lema:

Lema 1.1. *Neka je R komutativni prsten koji ima primitivni $n-$ti koren iz jedinice ω, gdje n i ω imaju multiplikativno inverzne elemente u R. Neka je $A, n \times n$ matrica čiji je element (i, j) jednak ω^{ij} za $0 \leq i, j < n$. Tada postoji A^{-1} i element (i, j) matrice A^{-1} je $(1/n)\omega^{-ij}$.*

Dokaz. Neka δ_{ij} bude jedan ako je $i = j$ i nula inače. Dovoljno je dokazati da ako je A^{-1} definisano kao gore, tada $AA^{-1} = I_n$, tj. $ij-$ti element u AA^{-1} je

$$\frac{1}{n}\sum_{k=0}^{n-1}\omega^{ik}\omega^{-kj} = \delta_{ij} \text{ za } 0 \leq i,j < n. \tag{1.3.1}$$

Ako je $i = j$, tada leva strana od (1.3.1) postaje

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

Za $i \neq j$, neka je $q = i - j$. Tada leva strana od (1.3.1) postaje

$$\frac{1}{n}\sum_{k=0}^{n-1}\omega^{qk}, \quad -n < q < n, q \neq 0.$$

Ako je $q > 0$ imaćemo

$$\frac{1}{n}\sum_{k=0}^{n-1}\omega^{qk} = 0,$$

jer je ω $n-$ti koren iz jedinice. Ako je $q < 0$, tada množenjem sa $\omega^{-q(n-1)}$, preuređujući red članova u sumi i zamenom q sa $-q$ dobijamo

$$\frac{1}{n}\sum_{k=0}^{n-1}\omega^{qk}, \quad 0 < q < n,$$

koje ponovo ima vrednost nula budući da je ω $n-$ti koren iz jedinice. Jednakost (1.3.1) sledi neposredno □

Vektor $F^{-1}(a) = A^{-1}a$ čija $i-$ta, $0 \leq i < n$, komponenta, je

$$\frac{1}{n}\sum_{k=0}^{n-1}a_k\omega^{-ik}$$

se zove *inverzna diskretna Furijeova transformacija vektora* a. Očigledno inverzna transformacija, transformacije vektora a, je samo a, tj. $F^{-1}F(a) = a$.

Postoji tesna veza DFT sa evaluacijom (nalaženjm vrednosti polinoma u datim tačkama) i interpolacijom polinoma. Neka je

$$p(x) = \sum_{i=0}^{n-1}a_i x^i$$

polinom stepena $n-1$. Polinom može biti jedinstveno predstavljen na dva načina, ili sa listom svojih koeficijenata $a_0, a_1, ..., a_{n-1}$ ili sa listom svojih

vrednosti u n različitih tačaka $x_0, x_1, ..., x_{n-1}$. Postupak nalaženja koeficijenata polinoma kada je on zadat svojim vrednostima u $x_0, x_1, ..., x_{n-1}$ se naziva interpolacija. Dok obrnuti postupak se naziva evaluacija.

Računanje DFT vektora $a = [a_0, a_1, ..., a_n]^T$ je ekvivalentno pretvaranju reprezentacije pomoću koeficijenata polinoma $\sum_{i=0}^{n-1} a_i x^i$ u reprezentaciju pomoću vrednosti u tačkama $\omega^0, \omega^1, ..., \omega^{n-1}$. Slično, inverzna DFT je ekvivalentna interpolaciji polinoma koji je zadat vrednostima u $n-$tim korenima iz jedinice.

Svakako evaluaciju polinoma mogli smo vršiti u proizvoljnih n tačaka ali $n-$ti koreni iz jedinice su praktično najpogodniji.

Jedna od uobičajenih primena DFT je računanje konvolucije dva vektora. Neka su

$$a = [a_0, a_1, ..., a_{n-1}]^T \text{ i } b = [b_0, b_1, ..., b_{n-1}]^T$$

vektori kolone. *Konvolucija* vektora a i b, u oznaci $a \otimes b$, je vektor

$$c = [c_0, c_1, ..., c_{2n-1}]^T \text{ gdje je } c_i = \sum_{j=0}^{n-1} a_j b_{i-j}.$$

Uzimamo $a_k = b_k = 0$ ako je $k < 0$ ili $k \geq n$. Tako

$$c_0 = a_0 b_0, c_1 = a_0 b_1 + a_1 b_0, c_2 = a_0 b_2 + a_1 b_1 + a_2 b_0,$$

itd. Napomenimo da uzimamao $c_{2n-1} = 0$ radi simetričnosti.

Da bi motivisali konvoluciju, posmatrajmo ponovo reprezentaciju polinoma pomoću koeficijenata. Proizvod dva polinoma $(n-1)-$og stepena

$$p(x) = \sum_{i=0}^{n-1} a_i x^i \text{ i } q(x) = \sum_{j=0}^{n-1} b_j x^j$$

je polinom $(2n-2)-$og stepena

$$p(x)q(x) = \sum_{i=0}^{2n-2} [\sum_{j=0}^{i} a_i b_{i-j}] x^i.$$

Uočimo da koeficijenti proizvoda polinoma su tačno komponente konvolucije vektora koeficijenata

$$a = [a_0, a_1, ..., a_{n-1}]^T \text{ i } b = [b_0, b_1, ..., b_{n-1}]^T$$

polaznih polinoma, ako mi zanemarimo c_{2n-1}, koje je nula.

Ako su dva polinoma stepena $(n-1)$ predstavljeni pomoću svojih koeficijenata, tada da bi smo sračunali koeficijente polinoma koji je njihov proizvod, mi treba da nađemo konvoluciju dva vektora koeficijenata. S druge strane, ako su polinomi $p(x)$ i $q(x)$ zadati pomoću vrednosti u $n-$tim korenima iz jedinice, tada da bi smo izračunali vrednosti polinoma koji je njihov

proizvod, u $n-$tim korenima iz jedinice, mi treba da pomnožimo parove vrednosti u odgovarajućim $n-$tim korenima. Ovo nam govori da konvolucija dva vektora **a** i **b** je inverzna transformacija vektora koji se dobija množenjem odgovarajućih komponenti transformacija datih vektora. Simbolički, $a \otimes b = F^{-1}(F(a) \cdot F(b))$. Tj. konvolucija može biti izračunata sa nalaženjem Furijeove transformacije, množenjem odgovarajućih parova i invertovanjem. Međutim, ovde se javlja jedan problem. Naime, proizvod dva polinoma stepena $(n-1)$ je polinom stepena $(2n-2)$ i za njegovu reprezentaciju nam trebaju vrednosti u $(2n-2)$ različite tačke. Ovaj problem je rešen u sledećoj teoremi sa posmatranjem $p(x)$ i $q(x)$ kao polinoma stepena $(2n-1)$, kod kojih su koeficijenti uz članove stepena većeg od $(n-1)$, nule (tj. tretiranjem polinoma stepena $(n-1)$ kao polinoma stepena $(2n-1)$).

Teorema 1.9 (Konvoluciona teorema) *Neka su*

$$a = [a_0, a_1, ..., a_{n-1}, 0, ..., 0]^T$$

i

$$b = [b_0, b_1, ..., b_{n-1}, 0, ..., 0]^T$$

vektori kolone dužine $2n$. *Neka su dalje*

$$F(a) = [a'_0, a'_1, ..., a'_{2n-1}]^T$$

i

$$F(b) = [b'_0, b'_1, ..., b'_{2n-1}]^T$$

njihove DFT. Tada je $a \otimes b = F^{-1}[F(a) \cdot F(b)]$.

Dokaz. Budući da je $a_i = b_i = 0$ za $n \leq i < 2n$, pisaćemo

$$a'_l = \sum_{j=0}^{n-1} a_j \omega^{lj} \text{ i } b'_l = \sum_{k=0}^{n-1} b_k \omega^{lk}.$$

Pa je

$$a'_l b'_l = \sum_{j=0}^{n-1} \sum_{k=0}^{n-1} a_j b_k \omega^{l(j+k)}. \tag{1.3.2}$$

Neka je $a \otimes b = [c_0, c_1, ..., c_{2n-1}]^T$ i $F(a \otimes b) = [c'_0, c'_1, ..., c'_{2n-1}]^T$. Kako je $c_p = \sum_{j=0}^{2n-1} a_j b_{p-j}$ mi imamo

$$c'_l = \sum_{p=0}^{2n-1} \sum_{j=0}^{2n-1} a_j b_{p-j} \omega^{lp} \tag{1.3.3}$$

Menjajući red sumiranja u (1.3.3) i zamenom $p-j$ sa k dobijamo

$$c'_l = \sum_{j=0}^{2n-1} \sum_{k=-j}^{2n-1-j} a_j b_k \omega^{l(j+k)} \tag{1.3.4}$$

Pošto je $b_k = 0$ za $k < 0$, mi možemo podići donju granicu od unutrašnje sume do $k = 0$. Na isti način, zbog $a_j = 0$ za $j \geq n$, mi možemo smanjiti gornju granicu spoljne sume na $n-1$. Sada gornja granica unutrašnje sume je najmanje n, kada je j najveće. Zato mi možemo zameniti gornju granicu sa $n-1$ budući da je $b_k = 0$ za $k \geq n$. Kada mi učinimo ove izmene (1.3.4) postaje identično sa (1.3.2), prema tome je $c'_l = a'_l b'_l$. Mi smo time pokazali da je $F(a \otimes b) = F(a)F(b)$ iz čega sledi da je $a \otimes b = F^{-1}(F(a)F(b))$ □

Konvolucija dva vektora dužine n je vektor dužine $2n$. Ovo zahteva dopunjavanje sa n nula vektora a i b u konvolucionoj teoremi. Da bi izbegli ovakvo dopunjavanje mi možemo upotrebljavati cikličnu konvoluciju.

Definicija 1.3. *Neka su*

$$a = [a_0, a_1, ..., a_{n-1}]^T \text{ i } b = [b_0, b_1, ..., b_{n-1}]^T$$

vektori dužine n. Pozitivnom cikličnom konvolucijom *vektora a i b nazivamo vektor* $c = [c_0, c_1, ..., c_{2n-1}]^T$ *gdje je*

$$c_i = \sum_{j=0}^{i} a_j b_{i-j} + \sum_{j=i+1}^{n-1} a_j b_{n+i-j}$$

Nega-ciklična konvolucija *je vektor* $d = [d_0, d_1, ..., d_{2n-1}]^T$ *gdje je*

$$d_i = \sum_{j=0}^{i} a_j b_{i-j} - \sum_{j=i+1}^{n-1} a_j b_{n+i-j}$$

Cikličke konvolucije koristićemo u Šenhage-Štrasenovom algoritmu, za brzo množenje velikih celih brojeva. Ako mi izvršimo evaluaciju dva polinoma stepena $(n-1)$ u $n-$tim korenima iz jedinice i zatim pomnožimo parove vrednosti u odgovarajućim korenima, to će nam dati n vrednosti kroz koje mi možemo interpolirati jedinstven polinom stepena $n-1$. Vektor koeficijenata ovog jedinstvenog polinoma je upravo pozitivna ciklična konvolucija vektora koeficijenata dva polazna polinoma.

Teorema 1.10. *Neka su*

$$a = [a_0, a_1, ..., a_{n-1}]^T \text{ i } b = [b_0, b_1, ..., b_{n-1}]^T$$

vektori dužine n. Neka ψ *bude primitivni* $2n-ti$ *koren iz jedinice i neka je* $\psi^2 = \omega$. *Pretpostavimo da* ψ, ω *i n imaju multiplikativno inverzne elemente.*

1. *Pozitivna ciklična konvolucija a i b je data sa* $F^{-1}(F(a)F(b))$.

2. *Neka je*

$$d = [d_0, d_1, ..., d_{n-1}]^T$$

nega-ciklična konvolucija vektora a i b. Neka je dalje

$$\hat{a} = [a_0, \psi a_1, ..., \psi^{n-1} a_{n-1}]^T,$$
$$\hat{b} = [b_0, \psi b_1, ..., \psi^{n-1} b_{n-1}]^T \ i$$
$$\hat{d} = [d_0, \psi d_1, ..., \psi^{n-1} d_{n-1}]^T.$$

Tada je $\hat{d} = F^{-1}(F(\hat{a})F(\hat{b}))$.

Dokaz je sličan dokazu Teoreme (1.9.) samo treba uočiti da je $\psi^n = -1$.

FFT algoritam

Direktno računanje b_j zahteva n množenja i $n-1$ sabiranje. Ovo daje $O(n^2)$ vremensku složenost za izračunavanje niza $\{b_0, b_1, ..., b_n\}$. Međutim postoji bolji algoritam. Neka je $n = 2^s$ za neki pozitivan ceo broj s. Izrazi za b_j mogu biti napisani kao

$$(1.3.5) \quad b_j = \sum_{m=0}^{2^s-1} a_m \omega^{mj} = \sum_{m=0}^{2^{s-1}-1} a_{2m} \omega^{2mj} + \sum_{m=0}^{2^{s-1}-1} a_{2m+1} \omega^{(2m+1)j}$$
$$= \sum_{m=0}^{2^{s-1}-1} a_{2m} (\omega^2)^{mj} + \omega^j \sum_{m=0}^{2^{s-1}-1} a_{2m+1} (\omega^2)^{mj}$$

za $j = 0, 1, ..., n-1$. Kako je $\omega^{n/2} = -1$, to (1.3.5) možemo zapisati kao

$$(1.3.6) \qquad b_j = \sum_{m=0}^{2^{s-1}-1} a_{2m} (\omega^2)^{mj} + \omega^j \sum_{m=0}^{2^{s-1}-1} a_{2m+1} (\omega^2)^{mj}$$

$$(1.3.7) \qquad b_{j+n/2} = \sum_{m=0}^{2^{s-1}-1} a_{2m} (\omega^2)^{mj} - \omega^j \sum_{m=0}^{2^{s-1}-1} a_{2m+1} (\omega^2)^{mj}$$

za $j = 0, 1, ..., 2^{s-1}-1$. Ovo vodi jednom rekurzivnom algoritmu za računanje b_j, jer svaka od dvije sume u (1.3.6) i (1.3.7) je DFT. Ovaj algoritam poznat je kao FFT (brza Furijeova transformacija) algoritam (vidi [1], [9], [12], [15]).

Algoritam $FFT(n, a_0, a_1, ..., a_{n-1}, \omega, \text{var} B)$;
Ulaz: $n = 2^k$(ceo broj), $a_0, a_1, ..., a_{n-1}$(niz na kojem se vrši DFT),
ω(primitivni n-ti koren)
Izlaz: B niz koji je DFT niza $\{ a_i \}$.

```
begin
    if n = 1 then B[0] = a_0
    else
        FFT(n/2, a_0, a_2, ..., a_{n-2}, ω^2, U);
        FFT(n/2, a_1, a_3, ..., a_{n-1}, ω^2, W);
        for j = 0 to n/2 - 1 do
            B[j] = U[j] + ω^j W[j];
            B[j + n/2] = U[j] - ω^j W[j];
end.
```

Kąko je predhodni (FFT) algoritam napisan na osnovu formula (1.3.6) i (1.3.7) to imamo

Teorema 1.11. *FFT algoritam računa DFT .*

Teorema 1.12. *FFT algoritam zahteva $O(n \log n)$ vremena.*

Dokaz. Označimo sa $T(n)$, složenost FFT algoritma, gdje je n veličina ulaznog niza. Tada dva rekurzivna poziva zahtevaju $2T(n/2)$ vremena, a *for* petlja $O(n)$ vremena. Pa imamo da je $T(n) = 2T(n/2) + O(n)$, iz čega na osnovu Teoreme (1.1.) dobijamo $T(n) = O(n \log n)$ □
Na osnovu Teorema (1.9.), (1.10.) i (1.12.) dobijamo sledeću posledicu.

Posledica 1.1. *Mi možemo sračunati $a \otimes b$, nega-cikličnu i pozitivnu cikličnu konvoluciju vektora a i b dužine n za $O(n \log n)$ vremena.*

FFT upotreba bit operacija

U mnogim primenama u kojima računamo konvoluciju mi zahtevamo tačan rezultat. Da bismo uprostili račun možemo upotrebiti DFT. Međutim, ako radimo u polju realnih brojeva, dolazi do greške, zbog aproksimacije realnih brojeva. Ovu grešku možemo izbeći sa izvođenjem računa u konačnom polju. Na primer, da bi smo našli $a \otimes b$, gdje je $a = [a_0, a_1, a_2, 0, 0]^T$, $b = [b_0, b_1, b_2, 0, 0]^T$ mi možemo uzeti 2 kao peti koren iz jedinice i izvršiti račun po modulu 31 (ovo ako smo sigurni da su komponente konvolucije manje od 30). Teškoća koja se javlja sa upotrebom konačnog polja je nalaženje odgovarajućeg polja koje ima $n-$ti koren iz jedinice. Zato ćemo mi upotrebljavati prsten R_m celih brojeva po modulu m, birajući m tako da R_m ima $n-$ti koren iz jedinice ω. No i dalje nije jednostavno za dato n, naći ω i m tako da je ω $n-$ti koren iz jedinice u R_m. Međutim, kad su n i $\omega > 1$ stepeni dvojke dokazaćemo (Teorema 1.13.) da konvoluciju možemo izračunati u prstenu celih brojeva po modulu $(\omega^{n/2} + 1)$ sa DFT, množenjem odgovarajućih komponenti i inverznim transformacijama. Prvo ćemo formulisati dva pomoćna tvrđenja (leme 1.2. i 1.3.). Smatraćemo da je $R = (S, +, *, 0, 1)$ komutativni prsten i $n = 2^k, k \geq 1$.

Lema 1.2. *Za sve $\omega \in S, \omega \neq 0$,*

$$\sum_{i=0}^{n-1} \omega^{ip} = \prod_{i=0}^{k-1} (1 + \omega^{2^i p}), \text{ za } 1 \leq p < n.$$

Dokaz. Dokaz izvodimo indukcijom po k. Baza indukcije, $k = 1$ je jasna. Sada, uočimo da je

$$(1.3.8) \quad \sum_{i=0}^{n-1} \omega^{ip} = \sum_{i=0}^{n/2-1} \omega^{2ip} + \sum_{i=0}^{n/2-1} \omega^{(2i+1)p} = (1 + \omega^p) \sum_{i=0}^{n/2-1} (\omega^2)^{ip}.$$

Koristeći indukcijsku hipotezu i zamenjujući ω^2 sa ω, dobijamo

$$\sum_{i=0}^{n/2-1} (\omega^2)^{ip} = \prod_{i=0}^{k-2} [1 + (\omega^2)^{2^i p}] = \prod_{i=1}^{k-1} [1 + \omega^{2^i p}]. \tag{1.3.9}$$

Pa zamenom (1.3.9) u desnu stranu od (1.3.8) dobijamo

$$\sum_{i=0}^{n-1} \omega^{ip} = (1 + \omega^p) \prod_{i=1}^{k-1} [1 + \omega^{2^i p}] = \prod_{i=0}^{k-1} [1 + \omega^{2^i p}] \ \square$$

Lema 1.3. *Neka je $m = \omega^{n/2} + 1$, gdje je $\omega \in S, \omega \neq 0$. Tada za $1 \leq p < n$ mi imamo $\sum_{i=0}^{n-1} \omega^{ip} \equiv 0$ po modulu m.*

Dokaz. Dovoljno je dokazati da je $1 + \omega^{2^j p} \equiv 0 (\mathrm{mod}\ m)$ za neko $j, 0 \leq j < k$ i primeniti Lemu (1.2.). Neka je $p = 2^s p'$, gdje je p' neparan. (Svakako $0 \leq s < k$.) Izaberimo j tako da je $j + s = k - 1$. Tada je $1 + \omega^{2^j p} = 1 + \omega^{2^{k-1} p'} = 1 + (m-1)^{p'}$. Ali $(m-1) \equiv -1 (\mathrm{mod}\ m)$ i p' je neparan, pa je $(m-1)^{p'} \equiv -1 (\mathrm{mod}\ m)$. Dakle, imamo da je $1 + \omega^{2^j p} \equiv 0 (\mathrm{mod}\ m)$ za $j = k - 1 - s$ $\square$

Teorema 1.13. *Neka su n i ω pozitivni stepeni broja 2 i neka je $m = \omega^{n/2} + 1$. Neka je dalje R_m prsten celih brojeva po modulu m. Tada u R_m, n i ω imaju multiplikativni inverzni po modulu m i ω je primitivni $n-$ti koren iz jedinice.*

Dokaz. Pošto su n i ω stepeni dvojke i m neparan, sledi da je m relativno prost sa n i ω. Zato n i ω imaju multiplikativno inverzne po modulu m. Kako važi $\omega \neq 1, \omega^n = \omega^{n/2} \omega^{n/2} \equiv (-1)(-1) = 1$ po modulu $\omega^{n/2} + 1$, i koristeći Lemu (1.3.) dobijamo da je ω primitivni $n-$ti koren iz jedinice u R_m $\square$

Značenje Teoreme (1.3.) je da konvoluciona teorema važi u prstenu celih brojeva po modulu $2^{n/2}+1$. Ako mi želimo izračunati konvoluciju dva vektora dužine n, sa celobrojnim komponentama koje su između 0 i $2^{n/2}$, tada mi možemo biti sigurni da ćemo dobiti tačan rezultat. Ako komponente nisu između 0 i $2^{n/2}$ tada su one korektne po modulu $2^{n/2} + 1$.

Pre nego što procenimo broj bit operacija za računanje konvolucije, razmotrimo broj bit operacija za računanje ostatka po modulu m, datog celog broja.

Neka je $m = \omega^p + 1$ za neki ceo broj p. Ako je a zapisan u bazi ω^p i notacijom kao niz od l blokova, svaki sa p cifara, tada a po modulu n može biti izračunat sa naizmeničnim sabiranjem i oduzimanjem l blokova do po p cifara.

Lema 1.4. *Neka je $m = \omega^p + 1$ i neka je $a = \sum_{i=0}^{l-1} a_i \omega^{pi}$, gdje su $0 \leq a_i < \omega^p$ za svako i. Tada je $a \equiv \sum_{i=0}^{l-1} a_i (-1)^i$ po modulu m.*

Dokaz. Uočimo da je $\omega^p \equiv -1$, iz čega dokaz sledi neposredno.

Napomenimo ako je l (broj blokova) u Lemi (1.4.) fiksiran, tada računanje ostatka broja a po modulu m može biti ostvareno sa $O_B(p \log \omega)$ bit operacija.

Lema (1.4.) daje jedan efikasan metod za računanje a po modulu m. Ona igra značajnu ulogu u sledećoj teoremi koja daje gornju granicu broja bit operacija potrebnih za računanje DFT i inverzne DFT.

Teorema 1.14. *Neka su ω i n stepeni broja 2 i $m = \omega^{n/2} + 1$. Neka je dalje $[a_0, a_1, ..., a_{n-1}]^T$ vektor sa celobrojnim komponentama, gdje je $0 \leq a_i < m$ za svako i. Tada DFT vektora $[a_0, a_1, ..., a_{n-1}]^T$ i njena inverzna transformacija mogu biti izračunate po modulu m za $O_B(n^2 \log n \log \omega)$ koraka.*

Dokaz. Upotrebićemo FFT algoritam, ali operacije (sabiranja, množenja) izvodimo po modulu m. Operacije se izvode $O(n \log n)$ puta. Lema (1.4.) nam kaže da sabiranje po modulu m zahteva $O_B(b)$ koraka, gdje je $b = ((n/2) \log \omega) + 1$. Množenje sa $\omega^p, 0 \leq p < n$ je ekvivalentno sa pomeranjem ulevo za $p \log \omega$ mesta, budući da je ω stepen dvojke. Dobijeni ceo broj nema više od $3b - 2$ bita, i tako sa Lemom 1.4. pomeranje i računanje ostatka zahteva $O_B(b)$ koraka. Tako DFT u direktnom smeru je vremenske složenosti $O_B(bn \log n)$ tj. $O_B(n^2 \log n \log \omega)$.

Inverzna transformacija zahteva množenje sa ω^{-p} i sa n^{-1}. Budući da je $\omega^p \omega^{n-p} \equiv 1 (\mathrm{mod}\ m)$, mi imamo $\omega^{n-p} \equiv \omega^{-p} (\mathrm{mod}\ m)$. Tako efekat množenja sa ω^{-p} može biti postignut množenjem sa ω^{n-p}. Poslednje je pomeranje ulevo za $(n-p) \log \omega$ mesta, a rezultujući ceo broj ima najviše $3b - 2$ bita. Opet ostatak može biti nađen za $O_B(b)$ koraka na osnovu Leme (1.4.). Na kraju, razmotrimo množenje sa n^{-1}. Ako je $n = 2^k$, tada mi pomerimo ulevo za $n \log \omega - k$ mesta, opet dobijamo broj od najviše $3b - 2$ bita, i sračunamo ostatak koristeći Lemu (1.4.). Dakle, inverzna DFT takođe zahteva $O_B(n^2 \log n \log \omega)$ koraka □

Posledica 1.2. *Neka $O_B(M(k))$ bude broj koraka potrebnih da se izračuna proizvod dva $k-$bitna cela broja. Neka su dalje a i b vektori dužine n sa celobrojnim komponentama u intervalu od 0 do ω^n, gdje su n i ω stepeni dvojke. Tada mi možemo izračunati $a \otimes b$ ili pozitivnu ili nega-cikličnu konvoluciju vektora a i b po modulu $\omega^n + 1$ za*

$$O_B(MAX[n^2 \log n \log \omega, nM(n \log \omega)])$$

vremena.

Prvi član u funkciji složenosti Posledice (1.2.) je vreme koje zahteva da bi se izvršila transformacija. Drugi član je cena izvršavanja $2n$ množenja $(n \log \omega + 1)-$bitnih celih brojeva. Napomenimo da ako za množenje brojeva iskoristimo Šenhage-Štrasenov algoritam, koji opisujemo kasnije, dobijamo da je $M(k) = k \log k \log \log k$. U ovom slučaju drugi član dominira prvi, zato nam je potrebno

$$O_B(n^2 \log n \log \log n \log \omega \log \log \omega \log \log \log \omega)$$

koraka da izvršimo konvoluciju.

2 Algoritmi za množenje velikih brojeva

Velike brojeve možemo pamtiti u nizu. Tačnije, veliki broj A možemo izraziti kao

$$A = \sum_{i=0}^{n} a_i B^i \tag{2.0.1}$$

za neku unapred zadatu bazu B. Pri tome je $0 \leq a_i \leq B-1$. Otuda je A određen sa $a_i, i = 0, ..., m$.

Iz zapisa (2.0.1) uočavamo sličnost između polinoma i velikih brojeva. Zbog toga se algoritmi za množenje polinoma mogu modifikovati do algoritama za množenje velikih brojeva.

Opišimo kratko ideje nekih algoritama (detaljnija objašnjenja se mogu naći u [1], [6], [8], [9], [10]).

2.1 Klasični algoritam

Zadatak: Dati su polinomi $p(x) = \sum_{i=0}^{m} p_i x^i$ i $q(x) = \sum_{i=0}^{m} q_i x^i$, naći $r(x) = p(x)q(x)$. Polinomi se zadaju svojim koeficijentima.

Koeficijente polinoma

$$r(x) = p(x)q(x) = \sum_{i=0}^{2m} r_i x^i$$

možemo računati kao

$$r_i = \sum_{j=0}^{i} p_j q_{i-j} \quad \text{za} \quad i \leq m$$

slično za $i > m$. Za računanje jednog r_i treba nam $i+1$ množenje, pa za računanje svih koeficijenata treba nam $O(m^2)$ množenja.

Da bi algoritam množio brojeve

$$P = \sum_{i=0}^{m} p_i B^i \quad \text{i} \quad Q = \sum_{i=0}^{m} q_i B^i,$$

treba još po završetku predhodnog algoritma dodati

$$r_{i+1} \leftarrow r_{i+1} + \lfloor r_i / B \rfloor, r_i \leftarrow r_i (\bmod B) \quad \text{za} \quad i = 0, 1, ..., m-1.$$

Ovim obezbeđujemo da je $0 \leq r_i \leq B-1$.

2.2 Karacuba-Hofmanov algoritam

Za množenje polinoma (velikih brojeva) možemo upotrebiti strategiju *podeli pa vladaj*. Naime polinom $p(x)$ možemo zapisati kao

$$p(x) = p'(x) + x^k p''(x), \text{ gdje je } k = \lfloor m/2 \rfloor.$$

Slično $q(x) = q'(x) + x^k q''(x)$. Pa je

$$r(x) = p(x)q(x) = p'(x)q'(x) + [p'(x)q''(x) + p''(x)q'(x)]x^k + p''(x)q''(x)x^{2k}$$

Time smo problem sveli na četiri množenja polinoma koji su duplo manjeg stepena od polaznih polinoma. Otuda je složenost

$$T(m) = 4T(m/2) + O(m) \text{ (na osnovu Teoreme 1.1.)} \Rightarrow T(m) = O(m^2).$$

Međutim, uočimo da množenje možemo izvršiti kao

$$r'(x) = p'(x)q'(x), \quad r''(x) = p''(x)q''(x)$$
$$r'''(x) = r'(x) + r''(x) - [p''(x) - p'(x)][q''(x) - q'(x)]$$
$$r(x) = p(x)q(x) = r'(x) + r'''(x)x^k + r''(x)x^{2k}$$

Sada smo problem sveli na tri množenja polinoma duplo manjeg stepena od polaznih polinoma. Pa je sada složenost

$T(m) = 3T(m/2) + O(m)$ iz čega na osnovu Teoreme 1.1. dobijamo

$$T(m) = O(m^{\log 3}) \text{ tj. } T(m) = O(m^{1.58}).$$

Uz male modifikacije ideju ovakvog algoritma možemo iskoristiti i za množenje velikih brojeva. Složenost je opet $O(m^{1.58})$.

Navedimo još neke rezultate (koji se mogu naći u [8]).

Teorema 2.1. *Za svako $\varepsilon > 0$ postoji konstanta $c(\varepsilon)$ i algoritam za množenje brojeva, za čiju složenost $T(n)$ važi*

$$T(n) < c(\varepsilon)n^{\varepsilon+1}.$$

Nedostatak ovakvog algoritma je što kada $\varepsilon \to 0$ konstanta $c(\varepsilon)$ raste ogromnom brzinom. Napomenimo još da postoje algoritmi (vidi [8]) čija je složenost

$$O(n2^{3.5\sqrt{\log_2 n}}) \text{ odnosno } O(n2^{\sqrt{2\log_2 n}}\log_2 n).$$

Štrasen je 1968. godine koristeći DFT i pamteći kompleksan broj ω (primitivni $n-$ti koren) s visokom tačnosti dobio do tada najbrži algoritam. Detalje vezane za algoritme bazirane na DFT opisujemo u poglavlju (2.4.). Šenhage i Štrasen su modifikovali ovakav algoritam i izbegli teškoće koje su imali pri radu sa kompleksnim brojevima. Ovaj algoritam je

$$O(n \log n \log\log n)$$

vremenske složenosti i njega ćemo detaljnije opisati u poglavlju (2.3).

2.3 Šenhage-Štrasenov algoritam

Šenhage i Štrasen su 1970. godine pronašli jedan elegantan algoritam za množenje velikih brojeva (vidi [1], [8]). Algoritam je $O_B(n \log n \log\log n)$ vremenske složenosti, ako množimo $n-$bitne brojeve.

Pre svega pogodno je zameniti n sa 2^n i naći proceduru koja množi 2^n bitne brojeve za $O_B(2^n n \log n)$ koraka. Prva ključna ideja ovog algoritma je množenje brojeva po modulu $2^{2^n}+1$. Ovim se ne umanjuje opštost, jer na primer, tačan rezultat množenja 32−bitnih brojeva možemo dobiti ako ih množimo po modulu $2^{64}+1$.

Pretpostavimo sada da je $N = 2^n$ i želimo da nađemo proizvod brojeva u i v po modulu 2^N+1. Ako je jedan od brojeva u ili v jednak $(-1) \equiv 2^N$ mi nemamo nikakvih teškoća, zato možemo smatrati da je $0 \leq u, v < 2^N$. Ako je na primer $u = -1$ tada je $uv \equiv (2^N + 1 - v) (\mathrm{mod}\ 2^N+1)$. Rastavimo naše N bitne brojeve na blokove. Neka je $k + l = n, K = 2^k, L = 2^l$,

$$u = U_{K-1}2^{(K-1)L} + \cdots + U_1 2^L + U_0$$

i

$$v = V_{K-1}2^{(K-1)L} + \cdots + V_1 2^L + V_0.$$

Odgovarajuće vrednosti za k i l odredićemo iz uslova koji će se pojaviti kasnije. Ispostavlja se da je potrebno da bude $k \leq l + 1$, a potom i da je najpogodnije uzeti $l = \lfloor n/2 \rfloor$.

Proizvod u i v je dat sa

$$uv = y_{2K-2}2^{(2K-2)L} + \cdots + y_1 2^L + y_0, \tag{2.3.1}$$

gdje je

$$y_i = \sum_{j=0}^{K-1} U_j V_{i-j}, 0 \leq i < 2K.$$

(Za $j < 0$ ili $j > K - 1$ uzimamo $U_j = V_j = 0$. Član y_{2K-1} je 0 i njega uzimamo samo radi simetričnosti.)

Proizvod u i v može biti izračunat upotrebom konvolucione teoreme. Množenje odgovarajućih parova DFT bi zahtevalo $2K$ množenja. Upotrebom ciklične konvolucije broj množenja smanjujemo na K. Ovo je razlog zbog koga se množenje izvodi po modulu $2^N + 1$. Kako je $KL = N$, mi imamo da je $2^{KL} \equiv -1 (\mathrm{mod}\ (2^N + 1))$. Tada iz (2.3.1) i Leme (1.4.)

$$uv \equiv (w_{K-1}2^{(K-1)L} + \cdots + w_1 2^L + w_0) (\mathrm{mod}\ (2^N + 1)),$$

gdje je $w_i = y_i - y_{K+i}, 0 \leq i < K$.

Budući da proizvod dva L bitna broja mora biti manji od 2^{2L} i kako su y_i i y_{K+i} sume $i+1$ i $K-(i+1)$ takvih proizvoda respektivno, to $w_i = y_i - y_{K+i}$ mora biti u intervalu $-(K-1-i)2^{2L} < w_i < (i+1)2^{2L}$. Tako postoji najviše

$K2^{2L}$ mogućih vrednosti koje w_i može uzeti. Ako možemo izračunati w_i po modulu $K2^{2L}$ onda možemo izračunati i $uv(\mathrm{mod}\ (2^N+1))$ sa $O(K\log(K2^{2L}))$ dodatnih koraka, sabiranjem w_i−ova uz odgovarajuća pomeranja.

Da bismo izračunali $w_i(\mathrm{mod}\ K2^{2L})$ mi računamo w_i dva puta, jedan put po modulu K, a drugi put po modulu $2^{2L}+1$. Ovo je druga ključna ideja ovog algoritma. Neka je $w_i' \equiv w_i(\mathrm{mod}\ K)$ i $w_i'' \equiv w_i(\mathrm{mod}\ (2^{2L}+1))$. Pošto je K stepen dvojke, a $2^{2L}+1$ je neparan to su K i $2^{2L}+1$ uzajamno prosti brojevi. Zato w_i može biti izračunato iz w_i' i w_i'' po formuli

$$w_i = (2^{2L}+1)((w_i' - w_i'')(\mathrm{mod}\ K)) + w_i''$$

i w_i je između $(K-1-i)2^{2L}$ i $(i+1)2^{2L}$. Rad potreban da se izračuna w_i iz w_i' i w_i'' je $O(L+\log K)$ za jedno w_i, pa za sve w_i ukupan rad iznosi $O(KL+K\log K)$ ili $O(N)$.

Za izračunavanje w_i' ne treba mnogo vremena, jer je k mnogo manje od $2L$. Možemo prvo izračunati $u_i' = u_i(\mathrm{mod}\ K)$, $v_i' = v_i(\mathrm{mod}\ K)$ i $y_i' = \sum_{j=0}^{2K-1} u_j' v_{i-j}'$ izračunati kako sledi. Nižimo u_i'−ove (v_i−ove) zajedno, razdvajajući ih sa $2\log K$ nula. Tako dobijamo brojeve

$$\hat{u} = \sum_{i=0}^{K-1} u_i' 2^{(3\log K)i} \quad \text{i} \quad \hat{v} = \sum_{i=0}^{K-1} v_i' 2^{(3\log K)i}.$$

Zatim izvršimo množenje brojeva $\hat{u}$ i $\hat{v}$ sa Karacuba-Hofmanovim algoritmom, što zahteva $O((3K\log K)^{1.58})$ koraka, tj. manje od $O(N)$ koraka. Vidimo da je $\hat{u}\hat{v} = \sum_{i=0}^{2K-1} y_i' 2^{(3\log K)i}$ i $y_i' < 2^{3\log K}$ pa y_i' možemo jednostavno dobiti iz proizvoda $\hat{u}\hat{v}$. Sada w_i' računamo kao $w_i' \equiv (y_i' - y_{K+i}')(\mathrm{mod}\ K)$.

Treća ključna ideja ovog algoritma je da se računanje w_i'' ostvaruje koristeći pojam ciklične konvolucije. Ovo obuhvata izvršavanje DFT, množenje odgovarajućih parova i inverznu DFT. Neka je $\psi = 2^{2^{l+1-k}}$ [3] (zbog ovoga se zahteva da je $k \le l+1$), $\omega = \psi^2 = 2^{4L/K}$ i $m = 2^{2L}+1$. Na osnovu Teoreme (1.13.) zaključujemo da ω i K imaju multiplikativno inverzne po modulu m i ω je primitivni K−ti koren iz jedinice. Tako je nega-ciklična konvolucija vektora $[u_0, \psi u_1, ..., \psi^{K-1}u_{K-1}]$ i $[v_0, \psi v_1, ..., \psi^{K-1}v_{K-1}]$

$$[y_0 - y_K, \psi(y_1 - y_{K+1}), ..., \psi^{K-1}(y_{K-1} - y_{2K-1})] \text{ po modulu } 2^{2L}+1,$$

gdje je $y_i = \sum_{j=0}^{K-1} u_j v_{i-j}$ za $0 \le i \le 2K-1$. Sada w_i'' dobijamo sa odgovarajućim pomeranjima. Kompletan algoritam je zapisan niže.

[2] Ako su p_1 i p_2 relativno prosti, $w \equiv q_1(\mathrm{mod}\ p_1)$, $w \equiv q_2(\mathrm{mod}\ p_2)$ i $0 \le w < p_1p_2$, tada je $w = p_2(p_2^{-1} \bmod p_1)(q_1 - q_2 \bmod p_1) + q_2$. Neka je $p_1 = K$ i $p_2 = 2^{2L}+1$. Kako je K stepen dvojke i $K \le 2^{2L}$, K deli 2^{2L} imamo da je 1 multiplikativni inverzni za $2^{2L}+1$ po modulu K.

[3] ψ biramo na ovaj način jer je $\omega^{K/2} = 2^{2L}$

Algoritam Šenhage-Štrasenov (Schönhage-Strassen)
Ulaz: Dva $N = 2^n$ bitna cela broja u i v.
Izlaz: $N + 1$ bitni proizvod u i v po modulu $2^N + 1$.
Metod: Ako je n malo, množimo u i v po modulu $2^N + 1$ sa nekim pogodnim algoritmom. Za veliko n izračunajmo $l = \lfloor n/2 \rfloor, k = n - l, L = 2^l, K = 2^k, \psi = 2^{2l+1-k}, \omega = \psi^2$. Izrazimo $u = \sum_{i=0}^{K-1} u_i 2^{Li}$ i $v = \sum_{i=0}^{K-1} v_i 2^{Li}$, gdje su u_i i v_i brojevi između 0 i $2^L - 1$ (tj. u_i su L bitni blokovi broja u, a v_i su L bitni blokovi broja v). Zatim

1. Izračunaj DFT, po modulu $2^{2L} + 1$, nizova

$$[u_0, \psi u_1, ..., \psi^{K-1} u_{K-1}] \text{ i } [v_0, \psi v_1, ..., \psi^{K-1} v_{K-1}]$$

   koristeći ω kao primitivni koren.

2. Izračunaj proizvode odgovarajućih parova DFT dobijenih u predhodnom koraku, i to po modulu $2^{2L} + 1$. Ovo ostvarujemo rekurzivnim pozivom Šenhage-Štrasenovog algoritma. Situacija kada je jedan od brojeva jednak 2^{2L} se tretira kao specijalan slučaj koji je jednostavan.

3. Izračunaj inverznu DFT po modulu $2^{2L}+1$ vektora proizvoda parova iz koraka 2. Rezultat ovoga je $[w_0, \psi w_1, ..., \psi^{K-1} w_{K-1}]$ po modulu $2^{2L}+1$, gdje je w_i $i-$ti član nega-ciklične konvolucije vektora $[u_0, u_1, ..., u_{K-1}]$ i $[v_0, v_1, ..., v_{K-1}]$. Izračunaj $w_i'' \equiv w_i (\text{mod } (2^{2L} + 1))$ množenjem $\psi^i w_i$ sa ψ^{-i} po modulu $2^{2L} + 1$.

4. a) Izračunaj $u_i' = u_i (\text{mod } K)$, $v_i' = v_i (\text{mod } K)$, za $0 \le i < K$.
   b) Izračunaj $\hat{u} = \sum_{i=0}^{K-1} u_i' 2^{(3 \log K) i}$ i $\hat{v} = \sum_{i=0}^{K-1} v_i' 2^{(3 \log K) i}$. Ovo ostvarujemo nižući $u_i'-$ove ($v_i'-$ove) zajedno sa $2 \log K$ nula između njih.
   c) Izračunaj proizvod $\hat{u}\hat{v}$ upotrebom Karacuba-Hofmanovog algoritma.
   d) Proizvod $\hat{u}\hat{v}$ je oblika $\sum_{i=0}^{2K-1} y_i' 2^{(3 \log K) i}$, gdje je $y_i' = \sum_{j=0}^{2K-1} u_j' v_{i-j}'$ i $y_i' < 2^{3 \log K}$. Izdvoji y_i' i izračunaj $w_i' \equiv (y_i' - y_{K+i}') (\text{mod } K)$, za $0 \le i < K$.

5. Izračunaj

$$w_i''' = (2^{2L} + 1)((w_i' - w_i'') \bmod K) + w_i'' \text{ i}$$

$$w_i = \begin{cases} w_i''' & \text{ako je } w_i''' < (i+1) 2^{2L} \\ w_i''' - K(2^{2L} + 1) & \text{ako je } w_i''' \ge (i+1) 2^{2L} \end{cases}$$

   za $0 \le i < K$.

6. Izračunaj $\sum_{i=0}^{K-1} w_i 2^{Li}$ po modulu $2^N + 1$. Ovo je željeni rezultat.

Teorema 2.2. *Algoritam Šenhage-Štrasenov računa uv po modulu* $2^N + 1$.

Dokaz. Sledi iz predhodnog izlaganja □

Teorema 2.3. *Vreme izvršavanja Šenhage-Štrasenovog algoritma je*

$$O_B(N \log N \log\log N)$$

koraka.

Dokaz. Označimo sa $m(N)$ vreme potrebno da se izmnože dva broja po modulu $2^N + 1$. Tada imamo da, DFT kao i inverznu DFT (tj. 1. i 3. korak) možemo ostvariti za $O_B(KL \log K)$ vremena (Teorema 1.14.). U drugom koraku imamo K množenja po modulu $2^{2L} + 1$, što možemo izvesti za $Km(2L)$ vremena. Dakle, prva tri koraka zahtevaju $O_B(KL \log K) + Km(2L)$ vremena. U četvrtom koraku konstruišemo brojeve $\hat{u}$ i $\hat{v}$ koji su dužine $3K \log K$ bita, a zatim ih množimo, što zahteva $O_B((3K \log K)^{1.58})$ vremena. Za dovoljno veliko K ovo vreme možemo zanemariti u odnosu na $O_B(KL \log K)$, jer $(3K \log K)^{1.58} < K^2$, a $KL \log K$ se ponaša kao $K^2 log K$. Takođe peti i šesti korak zahtevaju $O_B(N)$ vremena, pa mogu biti ignorisani.
Prema tome, dobijamo da je

$$m(N) = Km(2L) + O_B(KL \log K) \text{ tj. } m(2^n) = 2^k m(2^{l+1}) + O_B(2^k 2^l \log 2^k).$$

Ako predhodnu relaciju podelimo sa 2^n i zamenimo $m(2^n)/2^n$ sa $M(n)$ dobijamo

$$M(n) = 2M(l+1) + O(k).$$

Ova formula opravdava izbor k i l kako je to ranije predloženo. Naime, naš cilj je da $M(n)$ bude što je moguće manje. To ćemo postići ako l izaberemo što je moguće manje. Imajući ovo u vidu i uslove $l + k = n$, $k \leq l + 1$ dobijamo da l i k treba izabrati kao $k = \lceil n/2 \rceil$ i $l = \lfloor n/2 \rfloor$, što smo mi i uradili.

Za ovako odabrano k i l možemo naći konstantu c tako da je

$$M(n) \leq 2M(\lfloor (n-2)/2 \rfloor + 2) + cn \text{ za sve } n \geq 3.$$

Iterirajući predhodnu nejednakost dobijamo

$$M(n) \leq 2^j M(\lfloor (n-2)/2^j \rfloor + 2) + c(j(n-2) + 2^{j+1} - 2) \text{ za } j = 1, 2, ...$$

Birajući $j \approx \log_2 n$, mi vidimo da je $M(n) = O(n \log n)$. Pa je

$$m(2^n) = O_B(2^n n \log n) \text{ tj. } m(N) = O_B(N \log N \log\log N) \square$$

2.4 Diskretna težinska transformacija

Tekst u ovom poglavlju je preuzet iz rada [6] i čini nezavisnu celinu u odnosu na ostali tekst. Pojmovi i oznake su kao u radu [6] i važe samo u ovom poglavlju. Ovaj tekst navodimo jer algoritam baziran na težinskoj transformaciji je korišćen za ispitivanje karaktera broja F_{22}, a sem toga možemo da ga koristimo kao bazni korak Šenhage-Štrasenovog algoritma. Nismo se opredelili za paralelizaciju ovakvog algoritma jer on množi brojeve s izvesno greškom, pa da bi grešku držali pod kontrolom pravimo ograničenja na veličinu brojeva koje množimo.

Kako smo predhodno opisali množenje dva N bitna broja može biti ostvareno sa $O(N \log N \log\log N)$ bit operacija. U radu [6] uvodi se koncept diskretne težinske transformacije (DWT) koji u određenim situacijama implicitnu O konstantu smanjuje. Ovo se postiže sa izbegavanjem dopunjavanja brojeva nulama, čime se postiže smanjene dužine niza na koji se primenjuje FFT (radna dužina).

Težinska transformacija i konvolucija

Za ceo broj x koji u nekoj reprezentaciji ima cifre $x_0, x_1, ..., x_{N-1}$, definišemo signal $\mathbf{x}$ kao kolekciju cifara

$$\mathbf{x} = \{x_j : 0 \leq j < N\}. \tag{2.4.1}$$

Za skalar A mi definišemo proizvod skalara i signala kao

$$A\mathbf{x} = \{Ax_j\}. \tag{2.4.2}$$

Za signale $\mathbf{a}$, $\mathbf{x}$ mi definišemo proizvod signala sa signalom kao

$$\mathbf{ax} = \{a_j x_j\}, \tag{2.4.3}$$

i ako su svi elementi signala nenulti, mi definišemo inverzni signal sa

$$\mathbf{a}^{-1} = \{a_j^{-1}\}. \tag{2.4.4}$$

DWT se definiše po analogiji sa DFT. Za težinski signal $\mathbf{a}$ smatraćemo da ima nenulte komponente. Težinska transformacija signala $\mathbf{x}$ je signal $\mathbf{X}$ čije komponente su

$$X_k = \sum_{j=0}^{N-1} a_j x_j g^{-jk}, \tag{2.4.5}$$

gdje je g primitivni $N-$ti koren iz jedinice u odgovarajućem domenu. Inverzna DWT je

$$x_j = (Na_j)^{-1} \sum_{k=0}^{N-1} X_k g^{kj}. \tag{2.4.6}$$

Da bismo iskazali direktnu vezu između DWT i DFT i kraće zapisali (2.4.5) i (2.4.6) mi upotrebljavamo sledeću notaciju

$$\begin{aligned} \mathbf{X} &= DWT(N,\mathbf{a})\mathbf{x} = DFT(N)\mathbf{ax}, \\ \mathbf{x} &= DWT^{-1}(N,\mathbf{a})\mathbf{X} = \mathbf{a}^{-1}DFT^{-1}(N)\mathbf{X}. \end{aligned} \tag{2.4.7}$$

Jednostavno je uočiti da je DWT upravo DFT u slučaju kada je $\mathbf{a} = \mathbf{1}$, gdje $\mathbf{1}$ označava $\{1, 1, ..., 1\}$.

Ako su $\mathbf{x}$,$\mathbf{y}$ dva signala dužine N tada ćemo njihovu cikličnu konvoluciju označavati sa $\mathbf{x} * \mathbf{y}$. Njene komponente su

$$(\mathbf{x} * \mathbf{y})_n = \sum_{j+k=n(\mathrm{mod}N)} x_j y_j. \tag{2.4.8}$$

Ključne delove ove ciklične konvolucije možemo izdvojiti definisanjem, za $b = 0$ ili 1, konvolucije

$$(\mathbf{x} * \mathbf{y})_n^{(b)} = \sum_{j+k=bN+n} x_j y_j, \tag{2.4.9}$$

tako je

$$\mathbf{x} * \mathbf{y} = (\mathbf{x} * \mathbf{y})^{(0)} + (\mathbf{x} * \mathbf{y})^{(1)}. \tag{2.4.10}$$

Nega-cikličnu konvoluciju označavaćemo sa

$$\mathbf{x} \bullet \mathbf{y} = (\mathbf{x} * \mathbf{y})^{(0)} - (\mathbf{x} * \mathbf{y})^{(1)}. \tag{2.4.11}$$

Sada možemo definisati težinsku cikličnu konvoluciju. Pretpostavimo da je $\mathbf{a}$ nenulti težinski signal dužine N. Težinska ciklična konvolucija dva signala $\mathbf{x}$, $\mathbf{y}$ dužine N je

$$\mathbf{x} *^a \mathbf{y} = \mathbf{a}^{-1}((\mathbf{ax}) * (\mathbf{ay})). \tag{2.4.12}$$

U specijalnom slučaju kada je težinski signal generisan skalarom A,

$$a_j = A^j, \tag{2.4.13}$$

težinska konvolucija ima jednostavan oblik

$$\mathbf{x} *^a \mathbf{y} = (\mathbf{x} * \mathbf{y})^{(0)} + A^N(\mathbf{x} * \mathbf{y})^{(1)}. \tag{2.4.14}$$

Imajući u vidu definicije direktne i inverzne DWT (2.4.5) i (2.4.6), možemo formulisati odgovarajuću teoremu analognu klasičnoj konvolucionoj teoremi. Iz (2.4.5), (2.4.6) i definicije težinske konvolucije (2.4.12) nalazimo da je

$$\begin{aligned} (\mathbf{x} *^a \mathbf{y})_n &= (\mathbf{a}^{-1})_n N^{-1} \textstyle\sum_{k=0}^{N-1} X_k Y_k g^{kn} \\ &= DWT^{-1}(N,\mathbf{a})(\mathbf{XY}) \\ &= \mathbf{a}^{-1}DFT^{-1}(N)[(DFT(N)\mathbf{ax})(DFT(N)\mathbf{ay})]. \end{aligned} \tag{2.4.15}$$

Poslednja jednakost nam pokazuje kako možemo računati težinsku konvoluciju pomoću postojećeg FFT algoritma za DFT.

Množenje pomoću težinske konvolucije cifara

Sem standardne reprezentacije nenegativnog celog broj x

$$x = \sum_{j=0}^{N-1} x_j W^j \tag{2.4.16}$$

za neku fiksiranu bazu W, pomoću cifara x_j koje zadovoljavaju

$$0 \leq x_j < W \tag{2.4.17}$$

ponekad je pogodno koristiti balansiranu reprezentaciju, što pretpostavlja da je W parno i da je

$$-W/2 \leq x_j < W/2. \tag{2.4.18}$$

Balansirana reprezentacija je posebno pogodna kada se koristi FFT sa brojevima u pokretnom zarezu, jer doprinosi smanjenju greške koja se javlja zbog zaokruživanja. Napomenimo da postoje i drugi načini reprezentacije brojeva koje mi nećemo navoditi.

Kada je račun baziran na aritmetici u pokretnom zarezu, funkcije $\lfloor\cdot\rfloor$, $\lceil\cdot\rceil$ i $< \cdot >$ su značajne, jer različiti koraci algoritma zahtevaju celobrojne cifre u određenom smislu. Ove funkcije su definisane kako sledi. Za ceo broj n,

$$\lfloor n \rfloor = \lceil n \rceil = < n >= n. \tag{2.4.19}$$

Inače za $z = n + e$, gdje je n ceo broj a $0 < e < 1$,

$$\lfloor z \rfloor = n, \quad \lceil z \rceil = n + 1$$

$$< z >= \begin{cases} \lfloor z + 1/2 \rfloor & \text{ako je } z \geq 0, \\ \lceil z - 1/2 \rceil & \text{ako je } z < 0 \end{cases}$$

U opisu pseudo koda mi ćemo za kompleksan signal **z** označavati

$$< z >= \{< Re(z_j) > + i < Im(z_j) > : \ 0 \leq j < N\}. \tag{2.4.20}$$

Ovakva operacija se koristi da se izdvoje korektne celobrojne konvolucione vrednosti iz kompleksnog rezultata u pokretnom zarezu.

Sada navedimo i sam algoritam.

Algoritam W:
(težinski konvolucioni algoritam za množenje brojeva x,y)

(1) Izaberi odgovarajuću reprezentaciju za **x,y**, zajedno sa odgovarajućom dužinom N i težinski signal **a**.

(2) Izračunaj $\mathbf{X} = DWT(N, \mathbf{a})\mathbf{x}$, i $\mathbf{Y} = DWT(N, \mathbf{a})\mathbf{y}$.

(3) Izračunaj $\mathbf{Z} = \mathbf{XY}$.

(4) Izračunaj $\mathbf{z} = DWT^{-1}(N, \mathbf{a})\mathbf{Z}$. (Ovo je težinska konvolucija $\mathbf{x} *^{\mathbf{a}} \mathbf{y}$.)

(5) Primeni $\mathbf{z} =< \mathbf{z} >$ ako se ne upotrebljva celobrojni FFT.

(6) Prilagoditi cifre $\{z_n\}$ u reprezentaciju cifara koja je izabrana.

Različite vrste množenja (direktno, polinoma, po modulu Fermaovih brojeva, po modulu Merseneovih brojeva, itd.) će se razlikovati samo u odgovarajućem izboru u koraku (1), i prilagođavanju cifara u koraku (6). Koraci (2) i (4) zahtjevaju $O(N \log N)$ aritmetičkih (nad rečima) operacija, dok koraci (3), (5) i (6) zahtjevaju samo $O(N)$ aritmetičkih operacija.

Množenje pomoću FFT

Pretpostavimo da su nenegativni brojevi x, y prikazani u obliku (2.4.16) i pretpostavimo dalje da su nizovi cifara $\{x_j\}$, $\{y_j\}$ dopunjeni nulama, u smislu da je

(2.4.21) $$x_j = y_j = 0 \text{ za } j \geq N/2.$$

Zato imamo da je proizvod

(2.4.22) $$xy = \sum_{n=0}^{N-1} (x * y)_n W^n.$$

Dakle, u algoritmu $\mathbf{W}$ mi bismo počeli sa:
(1) Predstavi x, y u bazi W, pri čemu cifre dopunjavamo nulama tako da je $x_j = y_j = 0$ za $j \geq N/2$, gdje je N dužina niza na koji se primenjuje FFT, i izaberi $\mathbf{a} = \mathbf{1}$.

Nove cifre $z_n = (\mathbf{x} * \mathbf{y})_n$ mogu da se ne slažu sa željenom reprezentacijom, tako da u koraku (6) algoritma $\mathbf{W}$ je potrebno izvršiti neke dodatne operacije (operacije prenosa) da bi se dobila željena reprezentacija. Ovo je posebno značajno kada rezultat koristimo za novo množenje, tj. kada se vraćamo ponovo u korak (1).

Kada se upotrebljava aritmetika u pokretnom zarezu, moramo voditi računa o korekciji greške. Zato nam je potrebno da za grešku važi ograničenje

(2.4.23) $$e_n = |Re(z_n) - (\mathbf{x} *^{\mathbf{a}} \mathbf{y})_n| < 1/2,$$

pa ćemo posle operacije $< \cdot >$ dobiti tačnu vrednost. Za aritmetiku u pokretnom zarezu kada se upotrebljava $Q-$bitna mantisa i pretpostavljajući da su mantise od $sin(\cdot)$ i $cos(\cdot)$ korektne do $Q-1$ bita, došlo se do zaključka da ograničenje oblika

(2.4.24) $$e_n < c 2^{-Q} W^2 N^{3/2} \log N$$

može važiti za univerzalnu konstantu $c \sim 1$ kada je upotrebljena standardna reprezentacija. U slučaju balansirane reprezentacije imamo

(2.4.25) $$e_n < c' 2^{-Q} W^2 N \log N.$$

Napomenimo da su ova ograničenja verovatnosna i bazirana na konačnom skupu eksperimenata.

Da bi se greška držala pod kontrolom, a imajući u vidu gornja ograničenja najrazumljivije je izabrati $W = 2^{16}$, a N se bira da ograničenja važe. Na primer na mašini sa 64-bitnom aritmetikom u pokretnom zarezu, veličinom reči $W = 2^{16}$ i balansiranom reprezetacijom možemo, na ovaj način, množiti brojeve koji imaju po 2^{24} bita.

Fermaovi brojevi i nega-ciklična konvolucija

Množenje po modulu F_m može biti izvedeno kako sledi. Izaberimo bazu W koja deli $F_m - 1$, recimo $W = 2^{2^m/N}$. Sada dopunjavanje nulama nije potrebno, jer je $W^N = -1 (\mathrm{mod}\ F_m)$, pa je

$$xy = \sum_{n=0}^{N-1} (x \bullet y)_n W^n (\mathrm{mod}\ F_m). \tag{2.4.26}$$

Drugim rečima, za cifre broja $xy (\mathrm{mod}\ F_m)$ možemo uzeti komponente nega-ciklične konvolucije signala x i y. Kao i pre, u direktnom FFT metodu, obično se zahteva redukcija ovih cifara u reprezentaciju koja je izabrana.

Koncept težinske transformacije i ovde dolazi do izražaja. Naime, neka je $a_j = A^j$, gdje je A $N-$ti koren iz -1. Možemo uzeti $A = e^{\pm \pi i/N}$, ako koristimo FFT sa aritmetikom u pokretnom zarezu. Prvi korak algoritma W u ovom slučaju je:

(1) Izaberimo bazu $W = 2^{2^m/N}$, gdje će N biti radna dužina i definišimo signal $\mathrm{a} = \{A^j\}$, gdje je $A = e^{-\pi i/N}$. Predstavimo x, y u bazi W, pri tome se dopunjavanje nulama vrši samo do N cifara.

Na ovaj način smanjujemo radnu dužinu za pola, ali dobijamo transformacije

$$X_k = \sum_{j=0}^{N-1} x_j e^{-\pi i j/N} e^{-2\pi i j k/N}, \tag{2.4.27}$$

koje su na kompleksnim signalima. Međutim, transformacije tipa (2.4.27) možemo ostvariti pomoću transformacija realnih signala dužine N, sa samo malim brojem dodatnih operacija koje ne utiču na asimptotsko vreme. Zaista, definišimo specijalne transformacije

$$X'_k = 2 \sum_{j=0}^{N-1} x_j \cos(\pi j/N) e^{-2\pi i j k/N}, \tag{2.4.28}$$

koje posle $O(N)$ množenja sa $\cos(\cdot)$, možemo ostvariti kao FFT realnih signala. Treba napomenuti da ova specijalna transformacija, strogo govoreći, nije težinska transformacija zato što težinski signal $\{\cos(\pi j/N)\}$ nije inverzibilan. Imajući u vidu identitet

$$X'_k = X_k + X_{k-1} \tag{2.4.29}$$

i simetriju

$$X_k = X^*_{N-k-1}, \tag{2.4.30}$$

mi možemo dobiti X_0 direktno iz (2.4.27), i tada upotrebiti rekurziju

$$X_1 = X'_1 - X_0,\ X_2 = X'_2 - X_1,\ \ldots, \tag{2.4.31}$$

tako da, zaista, DWT (2.4.27) može biti dobijena pomoću FFT nad realnim signalima i $O(N)$ dodatnih operacija. Zbog simetričnosti (2.4.30) mi treba da sračunamo samo X_k i Y_k za $0 \leq k < N/2$. Slično za kraj inverzne transformacije, mi možemo izračunati komponente nega-ciklične konvolucije

$$z_n = (\mathbf{x} * \mathbf{y})_n = e^{\pi i n/N} N^{-1} \sum_{k=0}^{N-1} X_k Y_k e^{2\pi i k n/N} \tag{2.4.32}$$

pomoću inverzne FFT s realnim rezultatima:

$$(2\cos(\pi n/N)) z_n = N^{-1} \sum_{k=0}^{N-1} (X_k Y_k + X_{k-1} Y_{k-1}) e^{2\pi i k n/N} \tag{2.4.33}$$

kada je $n \neq N/2$, a komponentu koja nedostaje računamo po formuli

$$z_{N/2} = -2N^{-1} \sum_{k=0}^{N/2-1} Im(X_k Y_k)(-1)^k. \tag{2.4.34}$$

Dakle, množenje po modulu Fermaovih brojeva možemo ostvariti bez dopunjavanja nulama, i time prepoloviti radnu dužinu, upotrebom FFT s realnim signalima i inverzne FFT s realnim rezultatima.

Moguće je radnu dužinu smanjiti do 1/4 dužine koju zahtjeva direktna primena FFT. U ovom slučaju moramo upotrebljavati FFT s kompleksnim signalima, tj. metod će biti primenjen u situacijama kada imamo vrlo brz kompleksan FFT algoritam. Kao i pre, neka fiksirana dužina reči bude $W = 2^{2^m}/N$, ali sada ceo broj x predstavljamo pomoću kompleksnog ekvivalenta,

$$x' = \sum_{j=0}^{N/2-1} (x_j + \mathbf{i} x_{j+N/2}) W^j, \tag{2.4.35}$$

sličnu reprezentaciju koristimo i za y. Definišimo novu radnu dužinu $N' = N/2$ i signal $\mathbf{a} = \{A^j\}$, gdje je $A^{N'} = \mathbf{i}$. Lako je pokazati da je

$$x'y' = \sum_{n=0}^{N'-1} (\mathbf{x}' *^a \mathbf{y}')_n W^n (\mathrm{mod}\ F_m), \tag{2.4.36}$$

gdje je težinska konvolucija (zbog predhodne definicije A)

$$\mathbf{x}' *^a \mathbf{y}' = (\mathbf{x}' * \mathbf{y}')^{(0)} + \mathbf{i}(\mathbf{x}' * \mathbf{y}')^{(1)}. \tag{2.4.37}$$

U ovom slučaju prvi korak algoritma **W** izgleda:

(1) Izaberi bazu $W = 2^{2^m/N}$, gdje je $N' = N/2$ radna dužina i definiši signal $\mathbf{a} = \{A^j\}$, gdje je $A = e^{-\pi i/(2N')}$. Predstavi x, y u bazi W sa po N' kompleksnih cifara, kao u (2.4.35), pri tome dopunjavanje nulama vršimo samo do N'.

Na kraju ponovimo da ovaj algoritam ima četiri puta manju radnu dužinu od direktnog FFT metoda, ali koristi kompleksnu FFT.

Pregled novih rezultata o primalnosti Fermaovih brojeva

Pepinov test je poslužio da se dokaže složenost brojeva

$$F_7, F_8, F_{10}, F_{13}, F_{14}, F_{20} \text{ i } F_{22}.$$

Iako su Selfridge i Hurwitz još 1963. godine dokazali da je F_{14} složen, i danas se prema nama dostupnim podacima ne zna nijedan njegov faktor. Da je F_{20} složen dokazali su 1987. godine Young i Buell (vidi [16]). Test je izveden na Cray-2, a provjeren pomoću Cray X-MP, da bi se otklonile sumnje u mogućnost greške. Kvadriranje je ostvareno pomoću algoritama baziranog na direktnom FFT metodu.

U radu [5] se izlaže rezultat dokaza da je F_{22} složen. Pri tome je kvadriranje (centralna operacija Pepinovog testa) izvedeno pomoću DWT algoritma. Korišćena je mašina Amdahl 5995M model 4550 mainframe, kao i mnoge radne stanice za kontrolu među rezultata. Sem dokaza da je F_{22} složen autori rada [5] su prikazali tablicu Selfridge-Hurwitzovih ostataka brojeva F_n, $n \leq 22$, i rešili karakter kofaktora F_n, $n \leq 22$ ne nalazeći novi prost faktor.

3 Paralelizacija algoritama

U okviru ove glave opisaćemo kako se može paralelizovati Šenhage-Štrasenov algoritam, prikazan u poglavlju (2.3). Ideja je da se posebno paralelizuje svaki od šest koraka navedenih u Šenhage-Štrasenovom algoritmu. Zato ćemo pre izlaganja paralelne verzije Šenhage-Štrasenovog algoritma navesti paralelne algoritme za nalaženje DFT (koraci 1. i 3.), i kako možemo ostvariti paralelizaciju koraka 4. i 6. Napomenimo da su koraci 2. i 5. jednostavni za paralelizaciju, pa ih nećemo posebno razmatrati.

Paralelizaciju izlažemo prvo na MIMD EREW modelu sa zajedničkom memorijom, koji ima $P = 2^p$ procesora. Pri tome pretpostavljamo da različiti procesori mogu istovremeno pristupiti različitim lokacijama (u daljem tekstu, ako nije drugačije naglašeno, smatraćemo da imamo ovakav model[4]). Na kraju ćemo analizirati koji bi nam realan model bio najpogodniji.

3.1 Paralelni FFT algoritmi

U literaturi se mogu naći razne varijante paralelnih algoritama (na raznim modelima) za nalaženje DFT (vidi na primer [2], [3], [11], [14]). Navedimo da se u [2] prikazuje FFT na mreži procesora, u [3] na MMX mašini (koja je jedna vrsta MIMD SM modela), u radu [14] na hiperkocki (ali se prikazuje paralelizacija tzv. Bluesteinovog FFT algoritma), dok se u radu [11] prikazuje FFT na CRAY-1 itd. Nas će interesovati samo slučaj kada je dužina niza na kome primenjujemo DFT stepen dvojke. U ovom slučaju FFT algoritam je veoma pogodan za implementaciju na hiperkocki (vidi [4], [13]).

Hiperkocka

Da bismo jednostavnije opisali arhitekturu hiperkocke (vidi [4]) pretpostavimo da su procesori numerisani od 0 do 2^n-1 za neku vrednost n. U hiperkocki dva procesora su povezana ako i samo ako se njihova binarna reprezentacija razlikuje u tačno jednoj bit poziciji. Tako indeksi susednih procesora se razlikuju za stepen broja 2 (ali obrnuto ne važi). Drugi način za opis hiperkocke je induktivan : 0−dimenzionalna hiperkocka je samo jedan procesor, a za k veće od 1 mi definišemo k−dimenzionalnu hiperkocku kao dve $(k-1)$−dimenzionalne hiperkocke sa vezama između odgovarajućih procesora u svakoj polovini.

Procesori hiperkocke imaju samo lokalnu memoriju. Ne postoji zajednička memorija. Procesori komuniciraju sa slanjem poruka kroz veze između procesora. Tako komunikacija između susednih procesora će biti brža od komunikacije između procesora koji moraju slati poruke kroz među-procesore. Svaki procesor čeka dok ne dobije odgovarajući podatak ili poruku na taj način se postiže sinhronizacija među procesorima.

[4] svaki realan model može biti simuliran na ovom modelu

Hiperkocka ima dva značajna svojstva. Prvo, broj veza između procesora je sasvim mali; u $k-$dimenzionalnoj hiperkocki svaki procesor ima k suseda i tako on ima k veza. Prema tome moguće je izgraditi vrlo velike hiperkocke a da nemamo problema sa neostvarljivim (nerealnim) brojem veza. Takode iz druge definicije hiperkocke se jednostavno vidi kako možemo udvostručiti veličinu hiperkocke sa dodavanjem druge hiperkocke iste veličine i povezivanjem procesora sa istim indeksima u obe hiperkocke. Drugo značajno svojstvo je što mnoge značajne topologije mogu biti umetnute u hiperkocku.

FFT i hiperkocka

Neka je niz $\{X_k\}$ DFT niza $\{x_n\}$ tj.

$$X_k = \sum_{n=0}^{N-1} x_n \omega_N^{nk} \text{ za } k = 0, ..., N-1, \tag{3.1.1}$$

gdje je ω_N primitivni $N-$ti koren iz jedinice u odgovarajućem domenu i $N = 2^r$.

Paralelni FFT može biti definisan jednostavno sa povlačenjem $P-1$ (gdje je $P = 2^d$ broj procesora, $d \leq r$) horizontalnih linija kroz leptire FFT dijagrama i izvodenjem računa između svake dvije linije u jednom procesoru. Ali ovakav algoritam ne proizvodi uredene transformacije i zahteva dva puta više računa i komunikacija od algoritma koji navodimo (vidi [13])[5].

FFT od jednog niza može biti ostvaren kao više transformacija kraćih nizova. Ako je $N = N_0 N_1$, tada oba X_k i x_k mogu biti preslikana (mapirana) u dvodimenzionalni niz. Neka je $n = i + jN_0$ i $k = l + mN_1$ i definišimo nizove $x(i,j) = x_n$ i $X(l,m) = X_k$. Iz (3.1.1) dobijamo

$$X(m,l) = \sum_{i=0}^{N_0-1} \omega_{N_0}^{im} \omega_N^{il} \sum_{j=0}^{N_1-1} x(i,j) \omega_{N_1}^{jl}. \tag{3.1.2}$$

$X(l,m)$, pa prema tome i X_k mogu biti izračunati u tri koraka:

A. Izračunaj N_0 transformacija dužine N_1 i pomnoži sa ω_N^{il}:

$$X^{(1)}(i,l) = \omega_N^{il} \sum_{j=0}^{N_1-1} x(i,j) \omega_{N_1}^{jl}. \tag{3.1.3}$$

B. Izračunaj N_1 transformacija dužine N_0:

$$X^{(2)}(m,l) = \sum_{i=0}^{N_0-1} X(i,l) \omega_{N_0}^{im}. \tag{3.1.4}$$

[5] Primedba važi kada je $d < r$

C. Transponuj

$$X(l,m) = X^{(2)}(m,l). \tag{3.1.5}$$

Predhodno mapiranje može biti uopšteno u $r-$dimenzione nizove

$$x(i_0,...,i_r) = x_n, \tag{3.1.6}$$

$$X(k_{r-1},...,k_0) = X_k, \tag{3.1.7}$$

gdje i_α i k_α mogu biti ili nula ili jedan. Ako izvršimo ponavljanje faktorizacije (3.1.2), dobićemo

$$X^{(s+1)}(i_0,...,i_{r-s-1},k_{r-s},...,k_{r-1}) = \omega_{2^{r-s-1}}^{ik_{r-s}} \sum_{i_{r-s}=0}^{1} X^{(s)}(i_0,...,i_{r-s},k_{r-s+1},...,k_{r-1})\omega_2^{i_{r-s}k_{r-s}}, \tag{3.1.8}$$

gdje je $i = i_{r-s-1}...i_0$ (binarno) i $s = 1,...,r$. Rekurentna relacija (3.1.8) se inicijalizuje sa $X^{(1)}(i_0,...,i_{r-1}) = x(i_0,...,i_{r-1})$. Faze od FFT za $N = 16$ su prikazane u *Tabeli 1.*

Tabela 1 Međufaze FFT-a za $N = 16$	Tabela 2 Primeri $i-$ciklusa za slučaj $d = 2$ i $r = 5$	Tabela 3 $i-$ciklusi za FFT sa dužinom $N = 16$ i $P = 8$ procesora
$X(i_0,i_1,i_2,i_3)$	$X(i_4i_3\|i_2i_1i_0)$	$X(i_3i_2i_1\|i_0)$
$X^{(2)}(i_0,i_1,i_2,k_3)$	$X(i_4i_3\|i_1i_2i_0)$	$X^{(2)}(i_0i_2i_1\|k_3)$
$X^{(3)}(i_0,i_1,k_2,k_3)$	$X(i_1i_3\|i_4i_2i_0)$	$X^{(3)}(i_0k_3i_1\|k_2)$
$X^{(4)}(i_0,k_1,k_2,k_3)$		$X^{(4)}(i_0k_3k_2\|k_1)$
$X^{(5)}(k_0,k_1,k_2,k_3)$		$X^{(5)}(k_1k_3k_2\|k_0)$

Da bi smo implementirali ovakav FFT na hiperkocki, mi ćemo prvo opisati klasu pridruživanja (mapiranja) niza u procesore nazvanu permutacija indeksa. Neka je $n = i_{r-1}...i_0$ (binarno), tada standardno mapiranje niza u procesore je ono u kojem element x_n ima lokalnu adresu $i_{k-d-1}...i_0$ u procesoru broj $i_{k-1}...i_{k-d}$. Mapiranje je određeno sa

$$x_n = x(i_{k-1},...,i_{k-d}|i_{k-d-1},...,i_0), \tag{3.1.9}$$

gdje se podela | uvodi da razdvoji lokalnu adresu (desno) od broja procesora (levo). Permutacija indeksa je mapiranje niza u procesore u kom se indeksi permutuju. Na primer, ako je $n = i_3i_2i_1i_0$ i $d = 2$, tada mapiranje u kojem x_n ima lokalnu adresu i_1i_3 u procesoru i_0i_2 je mapiranje permutacijom indeksa x_n u procesor koje je određeno sa $x(i_0i_2|i_1i_3)$.

Da bismo implementirali ili pretvarali permutacije indeksa jednu u drugu, koristićemo $i-$cikluse (kako je uvedeno u radu [13]), za opisivanje paralelnih komunikacija algoritama na hiperkocki. $i-$ciklus je jedna permutacija indeksa od x_n u kojoj najznačajnija cifra (pivot) od lokalne adrese se menja

sa drugom cifrom bilo u lokalnoj adresi bilo u broju procesora. Pivot se uvek nalazi desno od podele |. Primeri $i-$ciklusa za slučaj $d = 2$ i $r = 5$ su dati u *Tabeli 2*. $i-$ciklus koji odgovara prvom i drugom članu u *Tabeli 2* ne zahteva međuprocesorsku komunikaciju jer se broj procesora nije promenio. Međutim, $i-$ciklus koji odgovara drugom i trećem članu će zahtevati komunikaciju između procesora jer se broj procesora menja.

$i-$ciklus ima tri značajna svojstva:

(i) Ako $i-$ciklus zahteva komunikaciju između procesora tada je ona direktna na hiperkocki, jer se procesori razlikuju u samo jednom bitu.

(ii) Ako $i-$ciklus zahteva komunikaciju, tada su elementi poređani uzastopno, jer je pivot najznačajnija cifra u lokalnoj adresi. Dužina paketa je uvek $N/(2P)$, a paketi se razmenjuju između procesora. Tj. procesor prima paket od procesora kome šalje paket. Prema tome $i-$ciklus može biti izveden in-place (bez dodatnog prostora).

(iii) Bilo koja permutacija indeksa, uključujući transponovanje matrice, premeštanje cifara u obrnutom redosledu, i razna druga premeštanja mogu biti izvedena sa $1.5d$ $i-$ciklusa. Uređeni FFT mogu biti ostvareni sa $1.5d$ do $2d$ $i-$ciklusa zavisno od relativne veličine od d i N (vidi [13]).

Neuređeni FFT može biti ostvaren sa $d + 1$ $i-$ciklusa kako je prikazano za slučaj $N = 4$ i $d = 3$ u *Tabeli 3*.

Grejovi kodovi, FFT i hiperkocka

U ovom poglavlju mi dajemo kratak pregled iz rada [4] u kome se opisuje kako se Grejov kod može iskoristiti za izvođenje brze Furijeove transformacije na hiperkocki.

Da bismo definisali Grejov kod, posmatrajmo brojeve između 0 i $2^n - 1$ i njihovu binarnu reprezentaciju. Grejov kod se tada definiše kao permutacija brojeva između 0 i $2^n - 1$ tako da susedni elementi imaju tačno jedan bit razlike (tako uzastopni elementi se razlikuju za stepen broja 2). Ovde mi takođe imamo u vidu da se prvi i poslednji element razlikuju tačno u jednom bitu. Na primer, za $n = 2$ Grejovi kodovi su $\{0, 1, 3, 2\}$ i $\{0, 2, 3, 1\}$. Postoji mnogo načina da se generiše Grejov kod i oni su proučavani za različite primene.

Takozvani *binary reflected* Grejov kod se obično upotrebljava. On je generisan rekurzivno kako sledi:

$$G_1 = \{0, 1\}$$

i ako imamo Grejov kod

$$G_k = \{g_0, g_1, g_2, ..., g_{2^k-1}\}$$

onda sledeći Grejov kod je dat sa

$$G_{k+1} = \{0g_0, 0g_1, 0g_2, ..., 0g_{2^k-1}, 1g_{2^k-1}, 1g_{2^k-2}, ..., 1g_0\}.$$

Očigledno je da su svi brojevi u G_{k+1} različiti i susedi imaju tačno jedan bit razlike. Grejov kod generisan na ovaj način označavaćemo sa G_k.

Drugi jednostavan način da se generiše Grejov kod je sledeći. Definišimo H_1 kao G_1 i neka je

$$H_k = \{h_0, h_1, h_2, ..., h_{2^k-1}\}.$$

Sledeći Grejov kod definišemo kao

$$H_{k+1} = \{h_0 0, h_0 1, h_1 1, h_1 0, h_2 0, h_2 1, ...\}.$$

Iako su G_k i H_k formirani na različite načine, ipak važi sledeća teorema (vidi [4])

Teorema 3.1. *$G_k = H_k$ za sve pozitivne cele brojeve k.*

Takođe važe sledeća teorema i njena posledica (vidi [4])

Teorema 3.2. *Ako je $G = \{g_0, g_1, g_2, ...\}$* binary reflected *grejov kod, tada se g_j i g_{j+2^k} razlikuju u tačno 2 bita za $k \geq 1$.*

Posledica 3.1. *FFT nad podacima koji su raspoređeni saglasno binary reflected Grejovom kodu mogu biti ostvarene sa slanjem jedne poruke na daljinu jedan i preostalih poruka na daljinu dva. Ovo je najkraća moguća daljina.*

FFT granični slučaj

Ovde prikazujemo jedan algoritam, za slučaj kada se broj procesora poklapa sa dužinom niza na koji primenjujemo DFT. Algoritam je sličan sa algoritmom koji se dobija sa povlačenjen $P - 1$ linije kroz leptire FFT dijagrama, ali je zanimljiv dokaz korektnosti koji je analitički.

Pretpostavimo da želimo da računamo DFT niza dužine P, smeštenog u zajedničkoj memoriji. Kako u sekvencijalnom FFT algoritmu, opisanom u poglavlju (1.3), imamo dva nezavisna rekurzivna poziva, to jednu polovinu raspoloživih procesora možemo angažovati da izvršavaju prvi rekurzivni poziv, a drugu polovinu procesora na izvršavanje drugog rekurzivnog poziva. Kombinacije dobijenih vrednosti možemo takođe izvoditi paralelno jer su operacije u *for* petlji FFT algoritma međusobno nezavisne. Koristeći se ovakvom idejom dolazimo do željenog algoritma.

Pre nego što opišemo sam algoritam navešćemo neke pojmove i svojstva koji su nam neophodni pri dokazu korektnosti predloženog algoritma.

Na skupu $J_i = \{j \in \mathbf{N} : 0 \leq j < 2^i\}, i \in \mathbf{N}$[6] definišimo preslikavanje $t_i : J_i \longrightarrow J_i$ kako sledi: Ako je $j = \sum_{p=0}^{i-1} d_p 2^p$, gdje je $d_p \in \{0, 1\}$ tada je

[6]Sa N smo ovdje označili skup prirodnih brojeva

$t_i(j) = \sum_{p=0}^{i-1} d_{i-p-1} 2^p$[7].
Preslikavanja t_i imaju sledeća svojstva:

1. $t_k(s2^i + j) = t_i(j)2^{k-i} + t_{k-i}(s)$, za $0 < i < k, s \in J_{k-i}, j \in J_i$;
2. $t_k(0) = 0, t_1(1) = 1$;
3. $t_{p+1}(2s) = t_p(s)$, za $s \in J_p$;
4. $t_{p+1}(2s+1) = 2^p + t_p(s)$, za $s \in J_p$;
5. $t_i(t_i(j)) = j$ za $j \in J_i$.

Dokaz.

1. Kako je $s \in J_{k-i}, j \in J_i$, to možemo pisati $s = \sum_{p=0}^{k-i-1} d_p 2^p$ i $j = \sum_{p=0}^{i-1} c_p 2^p$, gdje su $c_p, d_p \in \{0, 1\}$. Sada imamo

$$m = s2^i + j = (\sum_{p=0}^{k-i-1} d_p 2^p)2^i + \sum_{p=0}^{i-1} c_p 2^p = \sum_{p=0}^{k-1} e_p 2^p$$

gdje je

$$e_p = \begin{cases} c_p & \text{ako je } 0 \le p < i \\ d_{p-i} & \text{ako je } i \le p < k \end{cases}$$

Pa je

$$t_k(m) = \sum_{p=0}^{k-1} e_{k-p-1} 2^p = \sum_{p=0}^{k-i-1} e_{k-p-1} 2^p + \sum_{p=k-i}^{k-1} e_{k-p-1} 2^p$$

$$= \sum_{p=0}^{k-i-1} d_{k-p-1-i} 2^p + 2^{k-i} (\sum_{p=0}^{i-1} e_{i-p-1} 2^p)$$

$$= \sum_{p=0}^{k-i-1} d_{k-i-1-p} 2^p + 2^{k-i} (\sum_{p=0}^{i-1} c_{i-1-p} 2^p)$$

$$= t_{k-i}(s) + 2^{k-i} t_i(j).$$

Što je trebalo dokazati.

2. Očigledno.
3. U formulu iz prve tačke uvrstimo $k = p+1, i = 1, j = 0$ dobijamo $t_{p+1}(2s) = t_1(0)2^{p+1-1} + t_{p+1-1}(s) = t_p(s)$.
4. U formulu iz prve tačke uvrstimo $k = p+1, i = 1, j = 1$ dobijamo $t_{p+1}(2s+1) = t_1(1)2^{p+1-1} + t_{p+1-1}(s) = 2^p + t_p(s)$.

[7] Ovo je preslikavanje $rev(j)$, (vidi [1])

5. Očigledno.

Treće i četvrto svojstvo ćemo koristiti za $p = k - n - 1$ tj. kao $t_{k-n}(2s) = t_{k-n-1}(s)$, $t_{k-n}(2s+1) = 2^{k-n-1} + t_{k-n-1}(s)$, za $s \in J_{k-n-1}$.

Sada sledi i paralelna verzija FFT algoritma koja se dobija razradom ranije navedene ideje.

Algoritam Paralelni $FFT(P, a_0, a_1, ..., a_{P-1}, \omega,$ var $B)$;

Ulaz: $P = 2^p (= 2^k)$(ceo broj), $a_0, a_1, ..., a_{P-1}$(niz na kojem se vrši DFT), ω (primitivni P-ti koren iz jedinice).

Izlaz: B niz koji je DFT niza $\{ a_i \}$.

begin

for $m = 0$ to $2^p - 1$ do in parallel $Program_m$;

end.

Pri tome $m-$ti procesor izvršava $Program_m$ koji je dat niže.

$Program_m$

begin

nađi binarni zapis broja $m = (d_{k-1}d_{k-2}...d_1d_0)_2$;

neka je $m_1 = (d_0d_1...d_{k-2}d_{k-1})_2$; /* tj. $m_1 = t_k(m)$; */

uzmi u $b[m] = a[m_1]$;

$j = 0$;

for $i = 0$ to $k - 1$ do

if $d_i = 1$ then

$l = m - 2^i$;

$b_m = \omega^{j2^{k-i-1}} b_m$;

razmeni b_m, b_l sa programom l ;

$b_m = b_l - b_m$;

$j = j + 2^i$;

else

$l = m + 2^i$;

razmeni b_m, b_l sa programom l ;

$b_m = b_m + b_l$;

end.

Da predhodni algoritam računa DFT sledi iz sledeće teoreme.

Teorema 3.3. *Posle $n-$tog prolaza programa kroz* for *petlju paralelnog FFT algoritma u nizu $\{b[s2^n + j]\}_{j=0}^{2^n-1}$ se nalazi DFT niza*

$$\{a[p2^{k-n} + t_{k-n}(s)]\}_{p=0}^{2^n-1} \text{ za svako } s,\ 0 \le s \le 2^{k-n} - 1$$

Specijalno za $n = k$ tj. na kraju paralelnog FFT algoritma u nizu $\{b[j]\}_{j=0}^{2^k-1}$ se nalazi DFT niza $\{a[p]\}_{p=0}^{2^k-1}$.

Dokaz.

Uočimo prvo da posle $n-$tog prolaza kroz *for* petlju $Programa_m$ gdje je $m = \sum_{p=0}^{k-1} d_p 2^p$, promenljiva j ima vrednost $j = \sum_{p=0}^{n-1} d_p 2^p$ tj. $m = s2^n + j$ za neko $s, 0 \leq s < 2^{k-n}$. Kao i da pri $n-$tom prolazu kroz *for* petlju brojač i ima vrednost $n-1$.

Dokaz teoreme izvodimo indukcijom po n.

Za $n = 0$ tj. pre prolaska kroz *for* petlju imamo jednočlane nizove $\{b[s]\}$ i pri tome je $b[s] = a[t_k(s)]$ za $s = 0, ..., 2^k - 1$; tj. tvrđenje važi.

Neka tvrđenje važi za neko n, $0 \leq n < k$ tj. u nizu $\{b[s2^n + j]\}_{j=0}^{2^n-1}$ se nalazi DFT niza $\{a[p2^{k-n} + t_{k-n}(s)]\}_{p=0}^{2^n-1}$ za svako s, $0 \leq s \leq 2^{k-n} - 1$ pa je

$$b[s2^n + j] = \sum_{p=0}^{2^n-1} a[p2^{k-n} + t_{k-n}(s)]\omega^{pj2^{k-n}}$$

Dokažimo da će se posle $n+1-$nog prolaza kroz *for* petlju u nizu $\{b[s2^{n+1} + j]\}_{j=0}^{2^{n+1}-1}$ nalaziti DFT niza $\{a[p2^{k-n-1} + t_{k-n-1}(s)]\}_{p=0}^{2^{n+1}-1}$ za svako s, $0 \leq s \leq 2^{k-n-1} - 1$. Zaista, niz $\{b[s2^{n+1} + j]\}_{j=0}^{2^{n+1}-1}$ se sastoji od podnizova $\{b[(2s)2^n + j]\}_{j=0}^{2^n-1}$ i $\{b[(2s+1)2^n + j]\}_{j=0}^{2^n-1}$ pa po indukcijskoj pretpostavci posle $n-$tog prolaza programa kroz *for* petlju u njima se nalazi DFT nizova $\{a[p2^{k-n} + t_{k-n}(2s)]\}_{p=0}^{2^n-1}$ i $\{a[p2^{k-n} + t_{k-n}(2s+1)]\}_{p=0}^{2^n-1}$.

S druge strane, podniz niza $\{a[p2^{k-n-1} + t_{k-n-1}(s)]\}_{p=0}^{2^{n+1}-1}$ s parnim indeksima je $\{a[p2^{k-n} + t_{k-n-1}(s)]\}_{p=0}^{2^n-1}$ a sa neparnim je $\{a[p2^{k-n} + 2^{k-n-1} + t_{k-n-1}(s)]\}_{p=0}^{2^n-1}$. Pa kako važi $t_{k-n}(2s) = t_{k-n-1}(s), t_{k-n}(2s+1) = 2^{k-n-1} + t_{k-n-1}(s)$, to je $\{a[p2^{k-n} + t_{k-n}(2s)]\}_{p=0}^{2^n-1}$ podniz s parnim indeksima, a $\{a[p2^{k-n} + t_{k-n}(2s+1)]\}_{p=0}^{2^n-1}$ podniz sa neparnim indeksima niza $\{a[p2^{k-n-1} + t_{k-n-1}(s)]\}_{p=0}^{2^{n+1}-1}$.

Pri $n+1$ prolazu kroz *for* petlju $Program_m$, $m = (2s+1)2^n + j$, $(0 \leq j < 2^n)$ izvrši :

$l = m - 2^n = (2s+1)2^n + j - 2^n = (2s)2^n + j$;

$b[(2s+1)2^n + j] = \omega^{j2^{k-n-1}} b[(2s+1)2^n + j]$;

razmeni $b[m], b[l]$ sa $l-$tim programom ;

$b[m] = b[l] - b[m]$;

jer je $d_n((2s+1)2^n + j) = 1$.

A $Program_m, m = (2s)2^n + j$, $(0 \leq j < 2^n)$ izvrši :

$l = m + 2^n = (2s)2^n + j + 2^n = (2s+1)2^n + j$;

razmeni b_m, b_l sa $l-$tim programom ;

$b_m = b_m + b_l$;

jer je $d_n((2s)2^n + j) = 0$.

Dakle, za $0 \leq j < 2^n$ dobijamo

$$\begin{aligned} b[(2s)2^n + j] &= b'[(2s)2^n + j] + \omega^{j2^{k-n-1}} b'[(2s+1)2^n + j] \\ &= \sum_{p=0}^{2^n-1} a[p2^{k-n} + t_{k-n}(2s)]\omega^{pj2^{k-n}} \end{aligned}$$

$$+\omega^{j2^{k-n-1}} \sum_{p=0}^{2^n-1} a[p2^{k-n} + t_{k-n}(2s+1)]\omega^{pj2^{k-n}}$$

$$= \sum_{p=0}^{2^n-1} a[p2^{k-n} + t_{k-n-1}(s)]\omega^{pj2^{k-n}}$$

$$+\omega^{j2^{k-n-1}} \sum_{p=0}^{2^n-1} a[p2^{k-n} + 2^{k-n-1} + t_{k-n-1}(s)]\omega^{pj2^{k-n}}$$

$$= \sum_{p=0}^{2^{n+1}-1} a[p2^{k-n-1} + t_{k-n-1}(s)]\omega^{pj2^{k-n-1}}.$$

$$b[(2s+1)2^n + j] = b'[(2s)2^n + j] - \omega^{j2^{k-n-1}} b'[(2s+1)2^n + j]$$

$$= \sum_{p=0}^{2^n-1} a[p2^{k-n} + t_{k-n}(2s)]\omega^{pj2^{k-n}}$$

$$-\omega^{j2^{k-n-1}} \sum_{p=0}^{2^n-1} a[p2^{k-n} + t_{k-n}(2s+1)]\omega^{pj2^{k-n}}$$

$$= \sum_{p=0}^{2^n-1} a[p2^{k-n} + t_{k-n-1}(s)]\omega^{p(j+2^n)2^{k-n}}$$

$$+\omega^{(j+2^{n+1})2^{k-n-1}} \sum_{p=0}^{2^n-1} a[p2^{k-n} + 2^{k-n-1} + t_{k-n-1}(s)]\omega^{p(j+2^n)2^{k-n}}$$

$$= \sum_{p=0}^{2^{n+1}-1} a[p2^{k-n-1} + t_{k-n-1}(s)]\omega^{p(j+2^n)2^{k-n-1}}$$

Odnosno, za $0 \leq j \leq 2^{n+1} - 1$ je

$$b[s2^{n+1} + j] = \sum_{p=0}^{2^{n+1}-1} a[p2^{k-n-1} + t_{k-n-1}(s)]\omega^{pj2^{k-n-1}}$$

gdje koristimo oznaku b' samo da ukažemo da je to stara vrednost tj. vrednost koja se dobija posle $n-$tog prolaza kroz *for* petlju. Ovo upravo znači da se u nizu $\{b[s2^{n+1} + j]\}_{j=0}^{2^{n+1}-1}$ nalazi DFT niza $\{a[p2^{k-n-1} + t_{k-n-1}(s)]\}_{p=0}^{2^{n+1}-1}$ □

Ocenimo složenost paralelnog FFT algoritma. Za to je dovoljno oceniti složenost jednog $Programa_m$, jer se programi izvršavaju paralelno.

Ako pretpostavimo da operacije unutar *for* petlje zahtevaju $O(1)$ vremena, tada kako se petlja izvršava k puta njena složenost je $O(k)$. Računanje binarnog zapisa broja m kao i broja m_1 takođe zahteva $O(k)$ vremena, pa je složenost paralelnog FFT algoritma $O(k)$. S druge strane ako pretpostavimo

da su a_i–ovi $L = 2^l$ bitni brojevi i da se operacije izvode u prstenu celih brojeva po modulu $2^{2L}+1$, a ω stepen dvojke mali broj (tj. ≤ 16), kao što je u našem slučaju, inaćemo da je složenost operacija unutar *for* petlje (sabiranje, množene sa ω^{stepen}) $O_B(L)$. Složenost će u ovom slučaju biti $O_B(kL)$. Primetimo da je složenost sekvencijalnog algoritma $O_B(k2^kL)$ pa je ubrzanje $c \cdot 2^k$ puta, za neku konstantu c, $0 < c \leq 1$ tj. ubrzanje je linearno po broju procesora.

3.2 Paralelizacija četvrtog koraka

Pri praktičnoj realizaciji Šenhage-Štrasenovog algoritma pogodno je nešto modifikovati četvrti korak. Naime, ako je broj K mali, na primer takav da proizvod $u_i'v_j'$ možemo pamtiti u jednom registru, ne moramo nizati u_i'–ove (v_i'–ove) sa nulama. Dovoljno je posmatrati u_i'–ove kao koeficijente jednog polinoma, a v_i'–ove kao koeficijente drugog polinoma, pa su y_i'–ovi koeficijenti polinoma koji je proizvod predhodnih. Za ovo možemo koristiti Karacuba-Hofmanov algoritam za množenje polinoma. Ako je pak K veliki broj, onda možemo nizati u_i'–ove (v_i'–ove) sa nulama, kako je to ranije opisano, a zatim rekurzivno primeniti Šenhage-Štrasenov algoritam koji je brži od Karacuba-Hofmanovog algoritma.

Recimo još i da pri nizanju u_i'–ove (v_i'–ove) sa nulama, ne moramo dodavati tačno $2\log K$ nula, već možemo dodavati $s \geq 2\log K$ nula, tako da broj $(s + \log K)$ bude stepen dvojke. Time bi broj dužine $(s + \log K)$ bita pamtili u celom broju registara, za K dovoljno veliko. Svakako ovo povećava dužine nizova za konstantan faktor ali olakšava nizanje. Ovo je samo prividno povećanje jer inače nizove $\hat{u}$ i $\hat{v}$ moramo dopunjavati nulama ako želimo da primenimo Šenhage-Štrasenov algoritam, tako da predhodno opisano nizanje ne povećava dužinu brojeva koje kvadriramo. Ovakvo nizanje je posebno pogodno pri paralelizaciji četvrtog koraka, jer svaki procesor može nezavisno od drugih puniti odgovarajuća polja broja $\hat{u}$ ($\hat{v}$).

Zbog specifičnosti brojeva $\hat{u}$ i $\hat{v}$ Šenhage-Štrasenov algoritam se nešto pojednostavljuje. Naime ne moramo izvršavati četvrti i peti korak, već za w_i možemo uzeti w_i''. Ovim ćemo izbeći dalja rekurzivna pozivanja paralelnog Šenhage-Štrasenovog algoritma.

Paralelnu verziju Šenhage-Štrasenovog algoritma, za ostvarivanje četvrtog koraka, možemo primeniti jedino ako je K dovoljno veliko. Pri tome nam nije potrebno svih K procesora, jer su brojevi $\hat{u}$ i $\hat{v}$ znatno manji od polaznih.

Jasno da je vreme koje zahteva ovaj korak zanemarljivo u odnosu na prva tri koraka, jer su brojevi $\hat{u}$ i $\hat{v}$ znatno manji od polaznih.

Pokažimo zbog čega možemo uprostiti Šenhage-Štrasenov algoritam kada množimo brojeve $\hat{u}$ i $\hat{v}$. Ideju prvo izlažemo u uopštenom obliku. Naime pretpostavimo da želimo da množimo brojeve u i v koji imaju "nešto" (kasnije će biti precizirano) manje do 2^{n-1} bita. Da bi dobili tačan rezultat množenje izvodimo po modulu $N = 2^{2^n}+1$. Odredimo K, L kao u Šenhage-

Štrasenovom algoritmu i pretpostavimo da brojevi u i v nemaju više od $S\frac{K}{2}$ bita. Zatim rastavimo ove brojeve na blokove

$$u = \sum_{i=0}^{K/2-1} U_i 2^{iS}, \ \ 0 \leq U_i < 2^S;$$

i

$$v = \sum_{i=0}^{K/2-1} V_i 2^{iS}, \ \ 0 \leq V_i < 2^S.$$

Proizvod brojeva u i v je dat sa:

$$uv = \sum_{i=0}^{K-1} y_i 2^{iS},$$

gdje je

$$y_i = \sum_{j=0}^{K/2-1} U_j V_{i-j}, \ 0 \leq i < K.$$

(Za $j < 0$ ili $j > K/2-1$ uzimamo $U_j = V_j = 0$. Član y_{K-1} je 0 i njega uzimamo samo radi simetričnosti.)

Uočimo da je $0 \leq y_i < \frac{K}{2}2^{2S}$, pa tačnu vrednost za y_i možemo naći računajući y_i po modulu $\frac{K}{2}2^{2S}$. Ako je $\frac{K}{2}2^{2S} < 2^{2L}+1$, onda y_i možemo naći koristeći cikličnu konvoluciju, birajući $m = 2^{2L}+1$ i $\psi = 2^{2L/K}$. Ovo se tačno poklapa sa prva tri koraka Šenhage-Štrasenovog algoritma primenjenog na brojeve

$$u1 = \sum_{i=0}^{K-1} U_i 2^{iL}, \ \ 0 \leq U_i < 2^S$$

i

$$v1 = \sum_{i=0}^{K-1} V_i 2^{iL}, \ \ 0 \leq V_i < 2^S$$

gdje uzimamo $U_i = V_i = 0$ za $i \geq K/2$. Uočimo da broj $u1$ ($v1$) možemo dobiti tako što između odgovarajućih blokova broja u (v) dopišemo $L-S$ nula. Dakle, ako važi uslov $\frac{K}{2}2^{2S} < 2^{2L}+1$, onda četvrti i peti korak Šenhage-Štrasenovog algoritma su suvišni.

Da bi uslov $\frac{K}{2}2^{2S} < 2^{2L}+1$ važio mora da važi $S \leq L - \frac{\log K - 1}{2}$ tj. $S \leq L - \lceil \frac{n+2}{4} \rceil$. Pa brojevi u i v mogu da imaju najviše $2^{n-1} - 2^{\lceil n/2 \rceil - 1}\lceil \frac{n+2}{4} \rceil$ bita. Na primer za $n = 21$ brojevi u i v treba da imaju manje od $2^{20} - 6 * 2^{10} = 1048576 - 6144 = 1042432$ bita.

Ovakva modifikacija nam može koristiti ako množenje izvodimo po modulu nekog broja koji je nešto manji od F_n. Tada, prvo treba sračunati tačan proizvod, tj. račun izvodimo po modulu F_{n+1}, pa onda nalazimo moduo po datom broju. No množenje po modulu F_{n+1} ostvarujemo sa uprošćenim (bez

koraka 4. i 5.) Šenhage-Štrasenovim algoritmom. Ovo je upravo slučaj kada koristimo Teoremu (1.8.) za utvrdivanje karaktera brojeva oblika $h2^n + 1$.

Posmatrajmo ponovo brojeve $\hat{u}$ i $\hat{v}$. Možemo ih zapisati kao

$$\hat{u} = \sum_{i=0}^{K-1} u_i' 2^{i(s+\log K)} = \sum_{i=0}^{K_1-1} \hat{U}_i 2^{iL_1}$$

i

$$\hat{v} = \sum_{i=0}^{K-1} v_i' 2^{i(s+\log K)} = \sum_{i=0}^{K_1-1} \hat{V}_i 2^{iL_1}$$

gdje su L_1 i K_1 nove vrednosti za L i K, za njh važi $L_1 * K_1 = 2K*(s+\log K)$. Dok je

$$\hat{U}_i = \sum_{j=0}^{m-1} u'_{m*i+j} 2^{j(s+\log K)},\ u_i' = 0 \text{ za } i \geq K$$

i

$$\hat{V}_i = \sum_{j=0}^{m-1} v'_{m*i+j} 2^{j(s+\log K)},\ v_i' = 0 \text{ za } i \geq K$$

gdje je m broj za koji važi $m(s+\log K) = L_1$. Vidimo da je $\hat{U}_i < 2^{L_1-s}$, $\hat{V}_i < 2^{L_1-s}$. U ranijim oznakama je $S = L_1 - s \leq L_1 - 2\log K < L_1 - \frac{\log K_1 - 1}{2}$, što znači da možemo izostaviti četvrti i peti korak.

Kako smo rekli ranije za paralelno izvršavanje četvrtog koraka nećemo koristiti sve raspoložive procesore, zato opišimo kako možemo predavati podatke tako da se nađu u odgovarajućim procesorima. Naime brojeve u'_{m*i+j}, $0 \leq j < m$ predajemo procesoru $m * i$, tako da on može da formira blok

$$\hat{U}_i = \sum_{j=0}^{m-1} u'_{m*i+j} 2^{j(s+\log K)}.$$

Slično za brojeve v_i'. Sada procesor $m * i$ se ponaša kao $i-$ti procesor pri paralelnom množenju brojeva $\hat{u}$ i $\hat{v}$.

3.3 Paralelizacija šestog koraka

Ovde opisujemo paralelizaciju 6-tog koraka Šenhage-Štrasenovog algoritma. Da bi paralelizovali ovaj korak koristimo se sledećom idejom (vidi [9]). Pretpostavimo da sabiramo dva broja $u = \sum_{i=0}^{K-1} u_i 2^{iL}$ i $v = \sum_{i=0}^{K-1} v_i 2^{iL}$, $0 \leq u_i, v_i < 2^L$; i da na raspolaganju imamo K procesora ($i-$ti procesor sadrži u_i i v_i). Tj. računamo $w = u + v = \sum_{i=0}^{K} w_i 2^{iL}$ (w_K može biti i 0). Tada $w_0 = u_0 + v_0 (\mathrm{mod}\ 2^L)$, $w_i = u_i + v_i + c_{i-1} (\mathrm{mod}\ 2^L)$, za $i = 1, 2, ..., K-1$; gdje je c_i prenos iz predhodnog zbira i može biti 0 ili 1. Da nebi čekali c_{i-1}, $i-$ti procesor može sračunati $w_i' = u_i + v_i (\mathrm{mod}\ 2^L)$ i $w_i'' = u_i + v_i + 1 (\mathrm{mod}\ 2^L)$, pa

naknadno odlučiti da li će za w_i uzeti w_i' ili w_i''. Na ovaj način sabiranje ostvarujemo paralelno bez čekanja da se sračuna predhodni zbir i odredi prenos. Da bi sračunali w_i', w_i'' treba nam $O(L)$ koraka, a da bi odlučili koji od njih da uzmemo treba nam najviše $O(K)$ koraka (na $i-$tom koraku odlučujemo za $i-$ti član).

Napomenimo da se račun može nešto i ubrzati, ali složenost se smanjuje samo za konstantan faktor (zato i koristimo predhodno iskazanu ideju koja je jednostavnija za razmatranje). Naime polovinu raspoloživih procesora angažujemo da računaju sumu nižih bitova, a polovinu procesora na računanje sume viših bitova, pri tome se računa i suma uvećana za jedan. Time se odluka, koju od suma da uzmemo, donosi za $O(\log K)$ koraka. Ideja ovakvog algoritma, u nešta izmenjenom obliku može se naći u [9].
Svakako, paralelizacija 6-tog koraka, koja sledi, je znatno komplikovanija ali u njenoj osnovi leži višestruka primena prethodno iskazane ideje.

Treba da izračunamo $w = \sum_{i=0}^{K-1} w_i 2^{Li}$ po modulu $2^N + 1$, gdje su $K = 2^k, L = 2^l$ ranije opisani i znamo da je $0 \leq |w_i| < 2^{3L}$. Pri tome $i-$tom procesoru je dodeljena (sadrzi je) vrednost w_i, a mi želimo da sadrži $(w)_i$ tako da je $0 \leq (w)_i < 2^L$ i $w = \sum_{i=0}^{K-1} (w)_i 2^{Li}$ po modulu $2^N + 1$, tj. da imamo cifre broja w. Da bi izračunali ovu sumu posebno ćemo sabirati pozitivne a posebno negativne članove, pa ih onda samo oduzeti kako bi dobili traženu sumu. Za $0 \leq i < K$ definišimo w_i^+ i w_i^- na sledeći način:

$$w_i^+ = \begin{cases} w_i & \text{ako je } w_i > 0 \\ 0 & \text{ako je } w_i \leq 0 \end{cases}$$

$$w_i^- = \begin{cases} -w_i & \text{ako je } w_i < 0 \\ 0 & \text{ako je } w_i \geq 0 \end{cases}$$

Treba izračunati $w^+ = \sum_{i=0}^{K-1} w_i^+ 2^{Li}$ po modulu $2^N + 1$, i $w^- = \sum_{i=0}^{K-1} w_i^- 2^{Li}$ po modulu $2^N + 1$. I na kraju $w = w^+ - w^- (\text{mod } 2^N + 1)$.

Sada je dovoljno ukazati kako se može paralelno izračunati w^+, jer se w^- računa na sličan način. Zapišimo $w_i^+, 0 \leq i < K$ kao $w_i^+ = c_i^+ 2^{2L} + b_i^+ 2^L + a_i^+$, pri tome je $0 \leq a_i^+, b_i^+, c_i^+ < 2^L$ jer je $0 \leq w_i^+ < 2^{3L}$. Zatim računamo $x_i^+ = c_{i-2}^+ + b_{i-1}^+ + a_i^+$, $y_i^+ = x_i^+ + 1$ i $z_i^+ = x_i^+ + 2$ za $0 \leq i \leq K+1$, gdje uzimamo $c_i^+, b_i^+, a_i^+ = 0$ za $i < 0$ ili $i \geq K$. Pri tome svako x_i^+, y_i^+, z_i^+ rastavljamo na dva dela, na primer $x_i^+ = px_i^+ 2^L + ox_i^+$, gdje je $0 \leq ox_i^+ < 2^L$. Očigledno je da će za "prenose" da važi $0 \leq px_i^+, py_i^+, pz_i^+ \leq 2$, zato i računamo tri sume u slučaju da prenos iz prethodnog zbira bude 0 ili 1 ili 2. Ostaje nam još da vidimo kakav je prenos iz predhodnog zbira i da se odlučimo da li da uzmemo ox_i^+ ili oy_i^+ ili oz_i^+. Dakle, $p_0^+ = 0$, a $(w^+)_0 = ox_0^+$, dok za $i > 0$ je:

$$p_i^+ = \begin{cases} px_i^+ & \text{ako je } p_{i-1} = 0 \\ py_i^+ & \text{ako je } p_{i-1} = 1 \\ pz_i^+ & \text{ako je } p_{i-1} = 2 \end{cases}$$

$$(w^+)_i = \begin{cases} ox_i^+ & \text{ako je } p_{i-1} = 0 \\ oy_i^+ & \text{ako je } p_{i-1} = 1 \\ oz_i^+ & \text{ako je } p_{i-1} = 2 \end{cases}$$

Pa je $w^+ = p_{K+1}^+ 2^{L(K+2)} + \sum_{i=0}^{K+1} (w^+)_i 2^{Li}$ po modulu $2^N + 1$, gdje je $0 \leq (w^+)_i < 2^L$. Kako je $LK = N$ tj. $2^{LK} = -1 (\text{mod } 2^N + 1)$ to imamo da je $w^+ = \sum_{i=0}^{K-1} (w^+)_i 2^{Li} - (p_{K+1}^+ 2^{2L} + (w^+)_{K+1} 2^L + (w^+)_K) (\text{mod } 2^N + 1)$. Da bi izbegli višestruki račun vrednost $(p_{K+1}^+ 2^{2L} + (w^+)_{K+1} 2^L + (w^+)_K)$ možemo dodavati na w^- umesto da je oduzimamo od w^+. Slično bi uradili za deo koji bi trebali da oduzimamo od w^-. Sa ovakvom modifikacijom bismo računali $x_0^+ = c_{K-2}^- + b_{K-1}^- + a_0^+$, $x_1^+ = c_{K-1}^- + b_0^+ + a_1^+$ i $x_i^+ = c_{i-2}^+ + b_{i-1}^+ + a_i^+$ za $2 \leq i \leq K-1$. Ostale formule ostaju iste samo što sve radimo za $0 \leq i \leq K-1$. Pa je $w^+ = p_{K-1}^+ 2^{LK} + \sum_{i=0}^{K-1} (w^+)_i 2^{Li} (\text{mod } 2^N + 1)$. Napomenimo da se ovako sračunato w^+ i w^- razlikuju od onih koje smo prvobitno definisali, ali i dalje je $w = w^+ - w^- (\text{mod } 2^N + 1)$.

Oduzimanje paralelizujemo na sličan način kao sabiranje. Tačnije $i-$ta komponenta je $e_i 2^L + (w^+)_i - (w^-)_i$ ili $e_i 2^L + (w^+)_i - (w^-)_i - 1$ u zavisnosti da li je bilo pozajmice u predhodnom oduzimanju. Vrednost e_i je pozajmica iz predhodnog razreda tj. jednaka 0 ako je preostala razlika pozitivna, inače je 1.

Algoritam koji vrši traženo sumiranje bi bio:
for $i = 0$ to $K-1$ do in paralel $Program_i$
Gdje $i-$ti procesor izvršava $Program_i$ koji ima sledeće korake
$Program_i$

1. Neka je $|w_i| = c_i 2^{2L} + b_i 2^L + a_i$, gdje je $0 \leq a_i, b_i, c_i < 2^L$. Ako je $w_i > 0$ tada uzmi $a_i^+ = a_i, b_i^+ = b_i, c_i^+ = c_i, a_i^- = 0, b_i^- = 0, c_i^- = 0$ inače uzmi $a_i^- = a_i, b_i^- = b_i, c_i^- = c_i, a_i^+ = 0, b_i^+ = 0, c_i^+ = 0$.

2. Pošalji c_i^+, c_i^- $(i+2)-$gom procesoru, b_i^+, b_i^- $(i+1)-$om procesoru i primi c_{i-2}^+, c_{i-2}^- od $(i-2)-$gog procesora, b_{i-1}^+, b_{i-1}^- od $(i-1)-$og procesora. Treba napomenuti da nulti procesor prima c_{K-2}^-, b_{K-1}^-, c_{K-2}^+, b_{K-1}^+ a prvi procesor prima c_{K-1}^-, c_{K-1}^+ od $(K-1)-$og i $(K-2)-$og procesora, a ovi im šalju te vrednosti.

3. Izračunaj $x_i^+ = c_{i-2}^+ + b_{i-1}^+ + a_i^+, x_i^- = c_{i-2}^- + b_{i-1}^- + a_i^-$. (Napomenimo da ako je $i = 0$ onda izračunamo $x_0^+ = c_{K-2}^- + b_{K-1}^- + a_0^+, x_0^- = c_{K-2}^+ + b_{K-1}^+ + a_0^-$, a ako je $i = 1$ onda $x_1^+ = c_{K-1}^- + b_0^+ + a_1^+, x_1^- = c_{K-1}^+ + b_0^- + a_1^-$.) A zatim izračunaj i $y_i^+ = x_i^+ + 1, z_i^+ = x_i^+ + 2, y_i^- = x_i^- + 1, z_i^- = x_i^- + 2$

4. Odredi vrednosti kao što su: $px_i^+, ox_i^+, \ldots$

5. Primi (čekaj) p_{i-1}^+, p_{i-1}^- od $(i-1)-$og procesora i odredi p_i^+, p_i^-. A zatim pošalji p_i^+, p_i^- $(i+1)-$vom procesoru. Ako je $i = 0$ onda se samo $p_0^+ = p_0^- = 0$ pošalje 1. procesoru.

6. Odluči se o $(w^+)_i$ i o $(w^-)_i$ tj. $(w^+)_0 = ox_0^+, (w^-)_0 = ox_0^-$ i

$$(w^+)_i = \begin{cases} ox_i^+ & \text{ako je } p_{i-1}^+ = 0 \\ oy_i^+ & \text{ako je } p_{i-1}^+ = 1 \\ oz_i^+ & \text{ako je } p_{i-1}^+ = 2, \end{cases}$$

$$(w^-)_i = \begin{cases} ox_i^- & \text{ako je } p_{i-1}^- = 0 \\ oy_i^- & \text{ako je } p_{i-1}^- = 1 \\ oz_i^- & \text{ako je } p_{i-1}^- = 2, \end{cases}$$

za $1 \leq i \leq K-1$.

7. Izračunaj $s_i' = (w^+)_i - (w^-)_i$, $e_i' = 0$ ako je $(w^+)_i \geq (w^-)_i$ inače $s_i' = 2^L + (w^+)_i - (w^-)_i$, $e_i' = 1$

8. Izračunaj $s_i'' = (w^+)_i - (w^-)_i - 1$, $e_i'' = 0$ ako je $((w^+)_i - 1) \geq (w^-)_i$ inače $s_i'' = 2^L + (w^+)_i - (w^-)_i - 1$, $e_i'' = 1$.

9. Izračunaj e_i i $(w')_i$ po formulama:
ako je $i = 0$ onda je $e_0 = e_0', (w')_0 = s_0'$ inače je

$$e_i = \begin{cases} e_i' & \text{ako je } e_{i-1} = 0 \\ e_i'' & \text{ako je } e_{i-1} = 1, \end{cases}$$

$$(w')_i = \begin{cases} s_i' & \text{ako je } e_{i-1} = 0 \\ s_i'' & \text{ako je } e_{i-1} = 1. \end{cases}$$

10. Ako je $i = K-1$ izračunaj $p = p_{K-1}^- - p_{K-1}^+ + e_{K-1}$ i predaj multom procesoru.

11. Izračunaj $(w)_i$ koje se dobija iz $(w')_i$ kada se na $\sum_{i=0}^{K-1}(w')_i 2^L$ doda p. Pa kako je $-2 \leq p \leq 3$ to je za $i > 0$ $(w)_i$ jednako $(w')_i$ eventualno umanjeno ili uvećano za jedan.

Nije teško proveriti da je $w = \sum_{i=0}^{K-1}(w)_i 2^L$ i da je $0 \leq (w)_i < 2^L$, tj. $(w)_i$ su cifre broja w.

Procenimo složenost ovakvog algoritma.
Kako se uglavnom radi sa kopiranjem, slanjem i sabiranjem $O(L)$ bitnih brojeva složenost većine koraka je $O(L)$. Recimo da u 5–om koraku i–ti procesor dobija vrednosti p_{i-1}^+, p_{i-1}^- od predhodnog procesora, vrši računanje p_i^+, p_i^- i predaje sledećem procesoru. Tj. procesor čeka da predhodne vrednosti budu sračunate. Kako su u pitanju $O(1)$ bitni brojevi to je potrebno $O(K)$ koraka da se sračuna i poslednja vrednost p_{K-1}^+, p_{K-1}^-. Otuda je složenost prvih pet koraka $O(K+L)$. Slično koraci 9 i 11 zahtevaju $O(L+K)$ vremena. Dakle, složenost čitavog računa je $O(L+K)$, pa kako su u našem slučaju L i K bliske vrednosti možemo reći da je složenost $O(L)$, tj. $O(\sqrt{N})$.

Ovde treba imati u vidu da je broj koraka manji od $(const) \cdot L$, i da je *const* broj zanemarljiv u odnosu na $\log N \log\log N$.

3.4 Paralelizacija Šenhage-Štrasenovog algoritma

Sada opišimo samu paralelizaciju Šenhage-Štrasenovog algoritma. Pretpostavimo da je $p = \lceil n/2 \rceil$ i n dovoljno veliko. Imamo:

Algoritam Paralelni Šenhage-Štrasenov

Ulaz: Dva $N = 2^n$ bitna cela broja u i v smeštena u zajedničkoj memoriji.[8]

Izlaz: $N + 1$ bitni proizvod u i v po modulu $2^N + 1$.

Metod: Izračunaj l, k, L, K, ψ, ω kao u Šenhage-Štrasenovom algoritmu. Takođe brojevi u i v su izraženi kao u Šenhage-Štrasenovom algoritmu, zbog toga možemo koristiti u_i—ove i v_i—ove. Sada:

1. Koristeći paralelni algoritam za računanje DFT, izračunaj DFT, po modulu $2^{2L} + 1$, nizova

$$[u_0, \psi u_1, ..., \psi^{K-1} u_{K-1}] \text{ i } [v_0, \psi v_1, ..., \psi^{K-1} v_{K-1}]$$

koristeći ω kao primitivni koren.

2. Izračunaj paralelno proizvode odgovarajućih parova DFT dobijenih u predhodnom koraku, i to po modulu $2^{2L} + 1$. Napomenimo da jedan procesor računa jedan proizvod. Ovo ostvarujemo pozivom Šenhage-Štrasenovog algoritma od strane svakog procesora.

3. Koristeći paralelni algoritam izračunaj inverznu DFT po modulu $2^{2L}+1$ vektora proizvoda parova iz koraka 2, i izdvoj w_i''.

4. Izračunaj paralelno w_i', kako je to predhodno opisano.
Napomenimo da ako imamo više od P procesora ovaj korak možemo izvoditi paralelno sa prva tri koraka.

5. Sada i—ti procesor na osnovu w_i' i w_i'' izračuna

$$w_i''' = (2^{2L} + 1)((w_i' - w_i'') \bmod K) + w_i'' \text{ i}$$

$$w_i = \begin{cases} w_i''' & \text{ako je } w_i''' < (i+1)2^{2L} \\ w_i''' - K(2^{2L}+1) & \text{ako je } w_i''' \geq (i+1)2^{2L} \end{cases}$$

za $0 \leq i < K$.

6. Izračunaj, koristeći paralelni algoritam $\sum_{i=0}^{K-1} w_i 2^{Li}$ po modulu $2^N + 1$. Ovo je željeni rezultat.

Kako je predhodni algoritam dobijen paralelizacijom Šenhage-Štrasenovog algoritma to imamo sledeću teoremu.

[8] Pretpostavka da su brojevi u zajedničkoj memoriji ne stvara bitna ograničenja, jer obično su veliki brojevi samo među rezultati nekih prethodnih operacija. To je slučaj i sa testovima primalnosti u kojima se operacija kvadriranja ponavlja veliki broj puta.

Teorema 3.4. *Paralelni Šenhage-Štrasenov algoritam računa uv po modulu* $2^N + 1$.

Pređimo sada na ocenjivanje složenosti paralelnog Šenhage-Štrasenovog algoritma. Tačnije dokažimo sledeću teoremu.

Teorema 3.5. *Pod navedenim pretpostavkama vreme izvršavanja paralelnog Šenhage-Štrasenovog algoritma na MIMD modelu sa zajedničkom memorijom i sa* $P = 2^p$ *procesora, gdje je* $p = \lceil n/2 \rceil$, *je*

$$O_B((N/P) \log N \log\log N)^9$$

koraka.

Dokaz. Paralelno izvršavanje DFT kao i inverzne DFT sa P procesora ubrzava korake 1. i 3. linearno po broju procesora (tj. $c \cdot P$ puta, za neku konstantu c, $0 < c \leq 1$) u odnosu na odgovarajuće korake u Šenhage-Štrasenovom algoritmu. U koraku 2. računamo paralelno (istovremeno) K $(= P)$ proizvoda pa je ubrzanje $P-$puta. Slično u koraku 5. računamo w_i paralelno pa je i korak 5. ubrzan P puta. Što se tiče 6.-tog koraka, složenost izloženog paralelnog algoritma je $O(K + L)$, što je zanemarljivo u odnosu na prva tri koraka. Takođe složenost paralelnog izvršavanja 4.-tog koraka je zanemarljiva u odnosu na prva tri koraka.

Dakle, svaki korak iz Šenhage-Štrasenovog algoritma smo ubrzali $c \cdot P$ puta ili ih možemo zanemariti (jer ne utiču bitno na složenost). Otuda je složenost paralelnog Šenhage-Štrasenovog algoritma

$$O_B((N/P) \log N \log\log N) \; \square$$

Kako je $p = \lceil n/2 \rceil$, $P = 2^p$ i $N = 2^n$ imamo da je $(N/P) = 2^{\lfloor n/2 \rfloor}$ tj. pod navedenim pretpostavkama složenost paralelnog Šenhage-Štrasenovog algoritma je $O_B(\sqrt{N} \log N \log\log N)$.

Sada razmatrajmo MIMD model sa zajedničkom memorijom koji ima $P = 2^p$ procesora, ali kada je $p < \lceil n/2 \rceil$ ili $p > \lceil n/2 \rceil$.

Ako je $p < \lceil n/2 \rceil$ što je u praksi najčešći slučaj, svaki procesor može simulirati rad $2^{\lceil n/2 \rceil - p}$ procesora pa je opet ubrzanje linearno po broju procesora tj. složenost je i dalje $O_B((N/P) \log N \log\log N)$. Specijalno ako je $2^{n-p} n \log n \geq 2^n$ tj. $n \log n \geq 2^p$ korake 4. i 6. ne moramo paralelizovati jer složenost se ne menja (tj. $\geq N$).

Naglasimo da je u predhodnim slučajevima ubrzanje linearno po broju procesora tj. $S(P) = c \cdot P$ pa je $E(N, P) = c$, gdje je c konstanta između 0 i 1.

Ako je pak p neznatno veće od $\lceil n/2 \rceil$ tada dodatne procesore možemo iskoristiti za paralelno izvršavanje koraka 4. sa prva tri koraka. S druge

[9] Tj. složenost je manja od $O(N)$, koje zahteva ulaz odnosno izlaz, zato je i uvedena pretpostavka da su brojevi u zajedničkoj memoriji.

strane ako je $p > \lceil n/2 \rceil$ dovoljno veliko onda se dodatni procesori mogu iskoristiti za paralelno izvršavanje unutrašnjih operacija DFT (tj. sabiranje, pomeranje), kao i paralelno izvršavanje množenja odgovarajućih parova u koraku 2. (što bi se ostvarivalo rekurzivnim pozivima paralelnog Šenhage-Štrasenovog algoritma) i time dalje smanjiti složenost algoritma. Ovaj slučaj bi trebalo dodatno proučiti.

Na kraju razmotrimo koji realan model bi bio najpogodniji za predložene algoritme. Tačnije razmotrimo kakve veze bi nam trebale između procesora da bi komunikacija bila brza.

Recimo prvo da drugi i peti korak ne zahtevaju nikakve komunikacije. Što se tiče DFT (direktne i inverzne) vidimo da $m-$ti procesor komunicira sa $l-$tim, pri čemu se binarni zapisi brojeva m i l razlikuju u jednoj bit poziciji. Otuda je najpogodniji model hiperkocka. Međutim, paralelni algoritam za DFT koji smo opisali u poglavlju (3.1) zahteva punjenje procesora u nešto isprepletanom redosledu (tj. $m-$ti procesor sadrži $a[t_k(m)]$). No ako ovaj procesor posmatramo kao da ima oznaku $t_k(m)$, onda će punjenje biti prirodno (tj. $m-$ti procesor sadrži $a[m]$), ali rezultati su u nešto isprepletanom redosledu (jer je $t_k(t_k(m)) = m$). Ovo nam ne stvara nikakve probleme, jer posle primene inverznih transformacija ponovo dobijamo da su podaci u procesorima raspoređeni na prirodan način. U šestom koraku vidimo da $i-$ti procesor uglavnom komunicira sa $(i+1)-$vim, jer se komunikacija $i-$tog procesora sa $(i+2)-$gim može ostvariti preko $(i+1)-$og procesora. Dakle, trebali bi hiperkocki dodati veze $(i, i+1)$, ako već ne postoje. Tačnije treba da dodamo veze $(2i-1, 2i)$, jer veze $(2i, 2i+1)$ već postoje. Ovim stepen svakog čvora (procesora) uvećavamo za jedan. Međutim, zbog četvrtog koraka morali bi imati i dodatne veze između procesora $(m(2i-1), m(2i))$[10].

Da bismo izbegli ove dodatne veze i koristili samo hiperkocku, možemo procesore puniti saglasno Grejovom kodu. Tada će procesori, koji sadrže $i-$ti i $(i+1)-$vi podatak, biti susedni, ali komunikacije kod DFT neće biti više direktne, već na daljini dva (Posledica 3.1).

Treća mogućnost je da izvršimo modifikaciju šestog koraka i da ga ostvarujemo na hiperkocki. Tačnije da umesto dodavanja veze $(i, i+1)$, mi ostvarujemo komunikaciju, između $i-$tog i $(i+1)-$og procesora, kroz među procesore. Ovim paralelnu verziju šestog koraka usporavamo za $O(\log K)$, ali je i dalje složenost ovog koraka zanemarljiva u odnosu na ostale.

Dakle, zaključujemo da Pepinov test na hiperkocki možemo ubrzati linearno po broju procesora.

3.5 Implementacija Pepinovog testa

U ovom poglavlju mi ćemo opisati implementaciju Pepinovog testa, koja je izvedena na transpjuterima $T800$, a u čijoj osnovi se nalaze predhodno izloženi algoritmi za množenje velikih brojeva. Ukazaćemo uglavnom samo

[10] m je broj opisan u poglavlju (3.2)

na neke bitne elemente implementacije koji se razlikuju ili dopunjavaju one predhodno opisane, jer će kompletni programi biti prikazani u prilogu.

Da bismo proverili da li je F_n prost ili ne, prema Teoremi (1.7.), treba da sračunamo $3^{2^{2^n}-1} (\text{mod } F_n)$. Tačnije treba izvršiti program

$a = 3$;
for $i = 1$ to $2^n - 1$ do
kvadriraj(a,n);

pri čemu procedura *kvadritaj(a,n)* vrši kvadriranje broja a po modulu F_n. Pošto su brojevi F_n uglavnom veliki, a uzastopnim kvadriranjem i broj a će brzo dostići veličinu F_n, opredelili smo se da za kvadriranje koristimo paralelni Šenhage-Štrasenov algoritam.

Opišimo prvo implementaciju Šenhage-Štrasenovog algoritma, jer ga pozivamo u okviru paralelnog Šenhage-Štrasenovog algoritma.

Veliki broj a pamtimo u nizu $\{a_i\}$, gdje su a_i cifre broja a u bazi B koja je stepen dvojke. U našem slučaju je $B = 2^{16}$ [11], a omogućeno je da se ova vrednost menja. Prvi član niza tj. a_0 je rezervisan za pamćenje broja cifara samog broja kao i znaka. Tj. $sign(a_0) = sign(a)$, a $abs(a_0) = \{\text{broj cifara od } a\}$. Dakle,

$$a = sign(a_0) \sum_{i=1}^{abs(a_0)} a_i B^{i-1}, \quad 0 \leq a_i < B.$$

Što se tiče same implementacije, opet grubo, možemo reći da se sastoji od šest koraka

1. Računavanje DFT,
2. rekurzivno kvadriranje komponenti,
3. inverzne DFT,
4. računanje w'_r,
5. računanje w_r,
6. računanje $\sum w_r 2^{rL}$.

FFT algoritam je implementiran nerekurzivno. Pri tome u izlaznom nizu (FFT algoritma) se nalaze vrednosti DFT ali komponente su u nešto ispreplетanom redosledu (vidi [8]). Ovo iz razloga što nam ovako isprepletani redosled odgovara kao ulaz u proceduru za računanje inverzne DFT. Množenje sa ω^{st} se svodi na pomeranje za $st * somg$ bita, gdje je $somg$ takav da je $\omega = 2^{somg}$.

Kao izlaz iz rekurzije koristili smo Karacuba-Hofmanov algoritam (vidi [10]). Svakako mnogo brži algoritam se dobija ako kao izlaz iz rekurzije koristimo algoritme bazirane na direktnom FFT metodu ili bazirane na DWT

[11] Kako bi proizvod dve cifre mogli da pamtimo u procesorskoj reči, koja je 32-bitna

(poglavlje 2.4). Međutim, tada bi trebali obezbediti posebne tehnike za kontrolu greške izazvane približnim računom ($\sin(\cdot), \cos(\cdot)$ znamo samo približno).

Pri implementaciji četvrtog koraka izvršena je mala modifikacija u odnosu na predlog u Šenhage-Štrasenovom algoritmu. Naime, kako je $K = 2^k$ mali broj (naprimer $K = 1024$ za $n = 20$) to y_i' možemo pamtiti u jednom registru. Zato ne moramo nizati u_i'–ove sa nulama, dovoljno je posmatrati u_i'–ove kao koeficijente jednog polinoma, pa su y_i'–ovi koeficijenti polinoma koji je kvadrat predhodnog. Za ovo možemo koristiti Karacuba-Hofmanov algoritam za množenje polinoma (vidi [12]). Ovim smo olakšali implementaciju i uštedeli vreme neophodno za nizanje u_i'–ova sa nulama.

U petom koraku w_r smo računali po formulama

$$w_r = \begin{cases} (2^{2L}+1)up + w_r'' & \text{za } up < r, \text{ ili } up = r, r + w_r'' < 2^{2L} \\ -[(2^{2L}+1)(2^k - up) - w_r''] & \text{za } up > r, \text{ ili } up = r, r + w_r'' \geq 2^{2L} \end{cases}$$

Gdje je $up = ((w_r' - w_r'') \bmod 2^k)$.

Dokaz. Kako je već opisano u Šenhage-Štrasenovom algoritmu w_r možemo računati kao

$$w_r''' = (2^{2L}+1)up + w_r'' \text{ i}$$

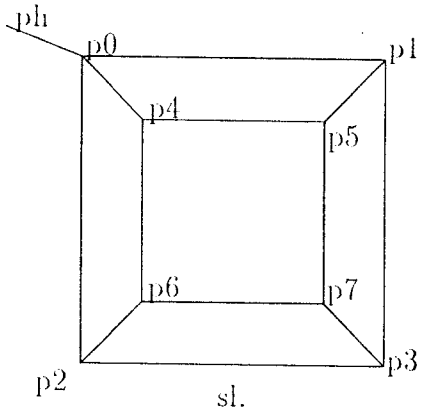
$$w_r = \begin{cases} w_r''' & \text{, ako je } w_r''' < (r+1)2^{2L} \\ w_r''' - 2^k(2^{2L}+1) & \text{, ako je } w_r''' \geq (r+1)2^{2L} \end{cases} \tag{3.5.1}$$

Međutim iz $w_r''' < (r+1)2^{2L}$ dobijamo $(2^{2L}+1)up + w_r'' < (r+1)2^{2L}$ odnosno $2^{2L}(up - r - 1) + up + w_r'' < 0$. Poslednja nejednakost će da važi ako je $up < r$, jer je $w_r'' \leq 2^{2L}$, a $up < 2^{2L}$. Ako je pak $up > r$ nejednakost očigledno ne važi jer su na levoj strani nenegativni brojevi. Dok u slučaju $up = r$ nejednakost postaje $-2^{2L} + up + w_r'' < 0$ tj. $r + w_r'' < 2^{2L}$. Dakle, $w_r''' < (r+1)2^{2L}$ ako i samo ako je $up < r$ ili $up = r$ i $r + w_r'' < 2^{2L}$. Iz čega neposredno dobijamo navedenu formulu.

Predhodna formula nam omogućava da nešto brže sračunamo w_r u odnosu na formule (3.6.1). Dovoljno je uporediti up i r, koji su "mali" brojevi, pa na osnovu toga odlučiti da li na w_r'' treba dodati $(2^{2L}+1)up$ ili oduzeti $(2^{2L}+1)(2^k - up)$.

Napomenimo još da u šestom koraku posebno sumiramo pozitivne a posebno negativne članove tako da novu vrednost dodajemo samo od odgovarajuće pozicije (slično kao što je opisano u poglavlju 3.3). Na kraju oduzmemo dobijene vrednosti po odgovarajućem modulu.

Za implementaciju Pepinovog testa (tj. paralelnog Šenhage-Štrasenovog algoritma) koristimo devet transpjutera povezanih kao na slici.



sl.

Procesor *ph* je glavni (domaćinski) procesor, ostali procesori služe kao pomoćni. Glavni program je

$a = 3$;

for $i = 1$ to $2^n - 1$ do

pkvadriraj(a,n);

Gdje se sada poziva procedura za paralelno kvadriranje broja a. U okviru ove procedure, a na osnovu veličine broja a, donosi se odluka da li treba uključiti i ostale procesore, ili prosto pozvati Šenhage-Štrasenov algoritam (izvršava ga samo procesor *ph*). Ako se donese odluka o paralelnom kvadriranju onda se vrednosti n i a predaju transpjuteru $p0$ i tada startuje paralelno računanje DFT. Pri tome je posao po procesorima podeljen na sledeći način: procesor *ph* izvršava četvrti korak, dok procesori $p0 - p7$ izvršavaju prva tri koraka (tj. četvrti korak izvršavamo paralelno sa prva tri). Zatim, procesor *ph* šalje dobijene vrednosti procesorima $p0 - p7$, koji izvrše peti i šesti korak. Rezultat se predaje procesoru *ph*. Napomenimo još da se paralelizacija šestog koraka ne vrši na način kako je to opisano u (3.3). Ovo iz razloga što imamo mali broj procesora pa šesti korak ne moramo paralelizovati jer ne utiče bitno na složenost celog algoritma. Pa se sabiranje ostvaruje tako što svaki od procesora izvrši sumiranje vrednosti koje sadrži, a zatim pola procesora preda dobijenu vrednost svojim predhodnicima, a onda ovi vrše sabiranje sa svojom vrednosti itd. sve dok se rezultat nenađe u $p0$. Potom $p0$ preda ovu vrednost procesoru *ph* i time se završava paralelno kvadriranje.

Uz pomoć ovog algoritma, koji oko četiri puta radi brže nego na jednom transpjuteru, potvrđeno je da su brojevi F_n složeni, za $n = 5, 6, ..., 15$. Takođe nađeni su Selfridge-Hurwitzovi ostaci koji se poklapaju sa rezultatima iz rada [5]. Tablicu dobijenih Selfridge-Hurwitzovih ostataka, kao i same programe dajemo u prilogu.

Na kraju, napišimo da je testiranje ispravnosti programa vršeno pomoću paketa UBASSIC. Takođe rezultati paralelnog Šenhage-Štrasenovog algoritma su proveravani pomoću sekvencijalnog Šenhage-Štrasenovog algoritma; a rezultati sekvencijalnog Šenhage-Štrasenovog algoritma su proveravani pomoću Karacuba-Hofmanovog algoritma.

Literatura

[1] Aho A.V., J.E. Hopcroft, and J.D. Ullman, *The Design and Analysis of Computer Algorithms*, Addison-Wesley, Reading, MA, 1974.

[2] Akl, Selim G., *The Design and Analysis of Parallel Algorithms*, London, Prentice-Hall International, cop.1989.

[3] Averbuch A., E. Gabber, B. Gordissky and Y. Medan, *A parallel FFT on an MIMD machine*, Parallel Computing 15 (1990) 61-74.

[4] Chamberlain R.M., *Gray codes, Fast Fourier Transforms and hypercubes*, Parallel Computing 6 (1988) 225-233.

[5] Crandall R., J.Doenias, C.Norrie, and J.Young, *The Twenty-Second Fermat Number is Composite*, Math. Comp. 60 (1995), 863-868.

[6] Crandall R. and B.Fagin, *Discrete weighted transforms and large-integer arithmetic*, Math. Comp. 62 (1994),305-324.

[7] Duncan, Ralph., *A Survey of Parallel Computer Architectures.* In: Survey & Tutorial series, 1990 February, 5-15.

[8] Knuth D.E., *The Art of Computer Programming, Volume 2 Seminumerical Algorithms*, Second edition, Addison-Wesley, Reading, MA, 1981.

[9] Manber U.,*Introduction to Algorithms A Creative Approach*, Addison-Wesley, Reading, MA, 1989.

[10] Riesel H., *Prime Numbers and Computer Methods for Factorization*, Birkhäuser Boston, Inc. 1985.

[11] Temperton C., *Implementation of a prime factor FFT algorithm on CRAY-1*, Parallel Computing 6 (1988) 99-108.

[12] Sedgewick R., *Algorithms*, Second edition, Addison-Wesley, Reading, MA, 1988.

[13] Swarztrauber P.N., *Multiprocessor FFTs*, Parallel Computing 5 (1987) 197-210.

[14] Swarztrauber P.N., *Bluestein's FFT for arbitrary N on the hypercube*, Parallel Computing 17 (1991) 607-617.

[15] Wilf H.S., *Algorithms and Complexity*, Prentice-Hall International, cop.1986.

[16] Young J. and D. Buell, *The twentieth Fermat number is composite*, Math. Comp. 50 (1988), 261-263.

Prilog

Definisanje ulazno izlaznih kanala za svaki procesor. Svaki kanal je oznaceen sa: P, zatim br. procesora sa kojim komunicira preko tog kanala i na kraju slovo I (za ulazni kanal) ili O (za izlazni kanal).

zp0.h

```
#define PHO ((Channel *) 0x8000000e)
#define PHI ((Channel *) 0x8000001e)
#define P1O ((Channel *) 0x80000000)
#define P2O ((Channel *) 0x80000004)
#define P4O ((Channel *) 0x80000008)
#define P1I ((Channel *) 0x80000010)
#define P2I ((Channel *) 0x80000014)
#define P4I ((Channel *) 0x80000018)
```

zp1.h

```
#define P0O ((Channel *) 0x80000000)
#define P3O ((Channel *) 0x80000004)
#define P5O ((Channel *) 0x80000008)
#define P0I ((Channel *) 0x80000010)
#define P3I ((Channel *) 0x80000014)
#define P5I ((Channel *) 0x80000018)
```

zp2.h

```
#define P3O ((Channel *) 0x80000000)
#define P0O ((Channel *) 0x80000004)
#define P6O ((Channel *) 0x80000008)
#define P3I ((Channel *) 0x80000010)
#define P0I ((Channel *) 0x80000014)
#define P6I ((Channel *) 0x80000018)
```

zp3.h

```
#define P2O ((Channel *) 0x80000000)
#define P1O ((Channel *) 0x80000004)
#define P7O ((Channel *) 0x80000008)
#define P2I ((Channel *) 0x80000010)
#define P1I ((Channel *) 0x80000014)
#define P7I ((Channel *) 0x80000018)
```

zp4.h

```
#define P5O ((Channel *) 0x80000000)
#define P6O ((Channel *) 0x80000004)
#define P0O ((Channel *) 0x80000008)
#define P5I ((Channel *) 0x80000010)
#define P6I ((Channel *) 0x80000014)
#define P0I ((Channel *) 0x80000018)
```

zp5.h

```
#define P4O ((Channel *) 0x80000000)
#define P7O ((Channel *) 0x80000004)
#define P1O ((Channel *) 0x80000008)
#define P4I ((Channel *) 0x80000010)
#define P7I ((Channel *) 0x80000014)
#define P1I ((Channel *) 0x80000018)
```

zp6.h

```
#define P7O ((Channel *) 0x80000000)
#define P4O ((Channel *) 0x80000004)
#define P2O ((Channel *) 0x80000008)
#define P7I ((Channel *) 0x80000010)
#define P4I ((Channel *) 0x80000014)
#define P2I ((Channel *) 0x80000018)
```

zp7.h

```
#define P6O ((Channel *) 0x80000000)
#define P5O ((Channel *) 0x80000004)
#define P3O ((Channel *) 0x80000008)
```

```
        #define P6I ((Channel *) 0x80000010)
        #define P5I ((Channel *) 0x80000014)
        #define P3I ((Channel *) 0x80000018)
zph.h
        #define P0O ((Channel *) 0x8000000c)
        #define P0I ((Channel *) 0x8000001c)


Nacini povezivanja programa tj. kako vrsimo linkovanje
m9ln.lnk
        INPUT unm,un1,un2,un3,un4
        LIB t8lib
m9ln0.lnk
        INPUT up0,un1,un2,un3,un4
        LIB t8lib
m9ln1.lnk
        INPUT up1,un1,un2,un3,un4
        LIB t8lib
m9ln2.lnk
        INPUT up2,un1,un2,un3,un4
        LIB t8lib
m9ln3.lnk
        INPUT up3,un1,un2,un3,un4
        LIB t8lib
m9ln4.lnk
        INPUT up4,un1,un2,un3,un4
        LIB t8lib
m9ln5.lnk
        INPUT up5,un1,un2,un3,un4
        LIB t8lib
m9ln6.lnk
        INPUT up6,un1,un2,un3,un4
        LIB t8lib
m9ln7.lnk
        INPUT up7,un1,un2,un3,un4
        LIB t8lib


NIF fajl u okviru koga ukazujemo na nacin punjenja
procesora odgovarajucim programom, kao i same veze
izmedju procesora.
m9net.nif
        1,unm,R0,0, , ,2;
        2,up0,R1,3,4,6,1;
        3,up1,S2,2,5,7, ;
        4,up2,R8,5,2,8, ;
        5,up3,R3,4,3,9, ;
        6,up4,R2,7,8,2, ;
        7,up5,S3,6,9,3, ;
        8,up6,R6,9,6,4, ;
        9,up7,R5,8,7,5, ;


/* UZAG.H :
        Zaglavlje u okviru kojeg definisemo zajednicke
        konstante i procedure koje koristimo.
*/
/* definicije zajednickih konstanti */
#define SBP 3 /* stepen broja procesora tj. 2^(SBP) = br. proc */
#define DUZN 33
#define TKX 8 /* x koordinata niza t */
#define TKY 129 /* y koordinata niza t */
```

```
#define DUZ_AP 257
#define DUZ_A 49152 /* velicine nizova koje kvadriramo ... */
#define DUZ_A1 24576
#define DUZ_UI 128 /* prva koordinata niza u[][] */
#define DUZ_UJ 256 /* druga koordinata niza u[][] */
#define DUZ_UJ1 33  /* druga koordinata niza u[][] */
#define SB 4 /* stepen baze */
#define DZB 16 /* duzina (tj. broj bita) baze */
#define BAZA 65536 /* vrednost baze */
/*
  Procedure definisane u UN4.C
*/
/*
  MULT: Izvrsava cetvrti korak Schonhage Strassenovog algoritma.
        Tj. nalazi konvoluciju niza y[] sa samim sobom po datom
      modulu. Zasniva se na Karatsuba & Ofmanovom algoritmu za
      mnozenje polinoma.
*/
extern void mult(int *y,int k);
extern void dodaj(int *,int *,int );/* Broju a dodaje broj  y*(BAZA)^(ip) */
/* SRACUNAJ: Ostvaruje peti korak Schonhage Strassenovog algoritma. */
extern void sracunaj(int ,int ,int , int ,int *);
/*
  Procedure definisane u UN3.C
*/
/* POMERI:
      Vrsi pomeranje datog broja ( b ) za pom mesta ulevo.
      Tj. mnozenje sa 2^pom . Sve to po modulu F_m.
*/
extern void pomeri(int *b,int pom,int m);
/* MODSUB:
     Nalazi razliku datih brojeva po modulu F_m.
*/
extern void modsub(int *a,int *b,int *c,int m);
/* PRETVORI:
     Datom negativnom broju dodaje F_m, kako bi dobio
     najmanju pozetivnu vrednost po modulu F_m.
*/
extern void pretvori(int *b,int m);
/* FFT: Nalazi FT datog niza.  */
extern void fft(int x[][DUZ_UJ1],int k,int m,int somg);
/* FFTI: Nalazi inverzne FT od datog niza. */
extern void ffti(int x[][DUZ_UJ],int k,int m,int somg);
/* FFTI1: Isto kao  FFTI samo sa manjim dimenzijama niza u... */
extern void ffti1(int x[][DUZ_UJ1],int k,int m,int somg);
/*
  Procedure definisane u UN2.C
*/

extern void inic();/* Vrsi inicijalizaciju nizova  sdva[] i t[]  */
/* KVADRIRAJ: Vrsi kvadriranje datog broja po modulu F_n.
         Procedura se zasniva na Schonhage Strassenovom algoritmu.
*/
extern void kvadriraj(int *a,int n);
/* SQUARE: Kvadrira dati broj po algoritmu Karatsuba & Ofman */
extern void square(int *p);
extern void add(int *a,int *b);/* Sabira date brojeve */
extern void sub(int *a,int *b);/* Nalazi (a-b), pretpostavlja a>b */
extern void moduo(int *a,int m);/* Nalazi (a mod F_m)  */
/*
  Procedure definisane u UN1.C
*/
extern void sendu(int *u,Channel *c);/* slanje niza(broja) u  preko kanala c */
```

```
extern void receiveu(int *u,Channel *c);/* prihvata niz u  preko kanala c */
/*
PREDAJ(...)
   iz niza  "a" izdvajaju se odgovarajuce komponente i salju preko kanala "c",
   pri tome duzine komponenti su "dnl" reci, a pocinje se od "ipoc" komponente
   i salje se svaka druga ili cetvrta... u zavisnosti od "dod".
*/
extern void predaj(int ipoc,int a0,int dnl,int dod,int *a,Channel *c);
/*PRIMI(...)
   radi suprotno od predaj(...) u prijemnom procesoru
*/
extern void primi(int ipoc,int a0,int dnl,int dod,int *a,Channel *c);
/*ABC1(...)
      pomocna procedura za ostvarivanje jedne kombinacije dva
      podniza u okviru FT, kako bi dobiji FT veceg niza,
      preciznije: a=a + 2^(pom)*b
*/
extern void abc1(int *a,int *b,int *c,int pom,int m);
/* ABC2(...):   kao abc1(...) samo b=a + 2^(pom)*b *\
extern void abc2(int *a,int *b,int *c,int pom,int m);
/*PP(...)
      vrsi izdvajanje odgovarajucih komponenti datog podniza
      nad kojim zatim primenjuje FT...
*/
extern void
pp(int k,int nmz,int dnl,int a0,int bp,int sfi,int m,int *a,int u[][DUZ_UJ]);



/* UN1.C */
#include <stdio.h>
#include <conc.h>
#include "uzag.h"
extern int sdva[21],t[TKX][TKY];

/*ABC1(...)
      pomocna procedura za ostvarivanje jedne kombinacije dva
      podniza u okviru FT, kako bi dobili FT veceg niza,
      preciznije: a=a + 2^(pom)*b, c=a - 2^(pom)*b
*/
void abc1(int *a,int *b,int *c,int pom,int m){
   int xp0,q,q1;
   pomeri(b,pom,m);
   modsub(a,b,c,m);
   if((a[0]==-1)||(b[0]==-1)){
      if(b[0]==-1){
         b[0]=1;
         modsub(a,b,a,m);
      }
      else{
         if(b[0]>0){
            xp0=b[0];
            a[1]=b[1]-1;q=1;
            while(a[q]<0){
               a[q]+=BAZA;q++;
               a[q]=b[q]-1;
            }
            for(q1=q+1;q1<=xp0;q1++)a[q1]=b[q1];
            while((xp0>0)&&(a[xp0]==0))xp0--;
            a[0]=xp0;
         }
      }
```

```
    }
    else{
       add(a,b);
       moduo(a,m);
    }
}/* abc1 */

/* ABC2(...):   kao abc1(...) samo b=a + 2^(pom)*b, c=a - 2^(pom)*b *\
void abc2(int *a,int *b,int *c,int pom,int m){
   int xp0,q,q1;
   pomeri(b,pom,m);
   modsub(a,b,c,m);
   if((a[0]==-1)||(b[0]==-1)){
      if(a[0]==-1){
         a[0]=1;
         modsub(b,a,b,m);
      }
      else{
         if(a[0]>0){
            xp0=a[0];
            b[1]=a[1]-1;q=1;
            while(b[q]<0){
               b[q]+=BAZA;q++;
               b[q]=a[q]-1;
            }
            for(q1=q+1;q1<=xp0;q1++)b[q1]=a[q1];
            while((xp0>0)&&(b[xp0]==0))xp0--;
            b[0]=xp0;
         }
      }
   }
   else{
      add(b,a);
      moduo(b,m);
   }
}/* abc2 */
void sendu(int *u,Channel *c){  /* slanje niza(broja) u  preko kanala c */
   int u0,i;
   u0=u[0];ChanOutInt(c,u0);
   if(u0<0)u0=(-u0);
   for(i=1;i<=u0;i++)ChanOutInt(c,u[i]);
}/* sendu */

void receiveu(int *u,Channel *c){  /* prihvata niz u  preko kanala c */
   int u0,i;
   u0=ChanInInt(c);u[0]=u0;
   if(u0<0)u0=(-u0);
   for(i=1;i<=u0;i++)u[i]=ChanInInt(c);
} /* receiveu */

/*
PREDAJ(...)
   iz niza  "a" izdvajaju se odgovarajuce komponente i salju preko kanala "c",
   pri tome duzine komponenti su "dnl" reci, a pocinje se od "ipoc" komponente
   i salje se svaka druga ili cetvrta... u zavisnosti od "dod".
*/
void predaj(int ipoc,int a0,int dnl,int dod,int *a,Channel *c){
   int i,j;
   i=ipoc;
   while((i+1)*dnl<=a0){
      for(j=1;j<=dnl;j++)ChanOutInt(c,a[i*dnl+j]);
      i+=dod;
   }
```

```
   if((i*dnl+1)<=a0){
      for(j=1;j<=a0-i*dnl;j++)ChanOutInt(c,a[i*dnl+j]);
   }
}/* predaj */

/*PRIMI(...)
   radi suprotno od predaj(...) u prijemnom procesoru
*/
void primi(int ipoc,int a0,int dnl,int dod,int *a,Channel *c){
   int i,j;
   i=0;
   while(((i*dod+ipoc+1)*dnl)<=a0){
      for(j=1;j<=dnl;j++)a[i*dnl+j]=ChanInInt(c);
      i++;
   }
   a[0]=i*dnl;
   if(((i*dod+ipoc)*dnl+1)<=a0){
      for(j=1;j<=a0-(i*dod+ipoc)*dnl;j++)a[i*dnl+j]=ChanInInt(c);
      a[0]+=(a0-(i*dod+ipoc)*dnl);
   }
}/* primi */

/*PP(...)   (skr. od pripremi)
      vrsi izdvajanje odgovarajucih komponenti datog podniza
      nad kojim zatim primenjuje FT...
*/
void
pp(int k,int mnz,int dnl,int a0,int bp,int sfi,int m,int *a,int u[][DUZ_UJ]){
   int i,j,it,nbm,u0;
   i=0;k-=SBP;
   while((mnz*i+1)*dnl<=a0){
      it=t[k][i];u0=dnl;
      while((u0>0)&&(a[mnz*i*dnl+u0]==0))u0--;
      for(j=1;j<=u0;j++)u[it][j]=a[mnz*i*dnl+j];
      u[it][0]=u0;i++;
   }
   if((mnz*i*dnl+1)<=a0){
      it=t[k][i];u[it][0]=a0-mnz*i*dnl;
      for(j=1;j<=u[it][0];j++)u[it][j]=a[mnz*i*dnl+j];
      i++;
   }
   nbm=sdva[k];
   for(j=i;j<nbm;j++){
      it=t[k][j];
      u[it][0]=0;
   }
   for(i=0;i<nbm;i++){
      it=t[k][i];
      pomeri(&u[it][0],(sdva[SBP]*i+bp)*sfi,m);
   }
   ffti(u,k,m,2*sfi*sdva[SBP]);
}/* pripremi */



/* UN2.C */
#include <stdio.h>
#include <conc.h>
#include "uzag.h"
extern int sdva[21],t[TKX][TKY];
int stri[9];
void inic(){  /* Vrsi inicijalizaciju nizova  sdva[] i t[]  */
```

```
    int i,p;
    p=1;
    for(i=0;i<8;i++){
       stri[i]=p;p*=3;
    }
    p=1;
    for(i=0;i<21;i++){
       sdva[i]=p;p*=2;
    }
    t[0][0]=0;
    for(i=0;i<=6;i++){
       for(p=0;p<sdva[i];p++){
          t[i+1][p]=2*t[i][p];
          t[i+1][p+sdva[i]]=t[i+1][p]+1;
       }
    }
}

/* KVADRIRAJ: Vrsi kvadriranje datog broja po modulu F_n.
          Procedura se zasniva na Schonhage Strassenovom algoritmu.
*/
void kvadriraj(int *a,int n){
   int l,k,sfi,somg,a0,i,izdnl,dnl,izdnl1;
   static int u[DUZ_UI][DUZ_UJ1],up[2*DUZ_UI],b[DUZ_AP];
   int mb,bm,j,ip,u0;
   if(a[0]<17){
      square(a);moduo(a,n);
   }
   else{
      while(a[0]<=sdva[n-2-SB])n--;
      l=n/2;k=n-l;
      sfi=sdva[l+1-k];somg=2*sfi;
      a0=a[0];i=0;izdnl=dnl=sdva[l-SB];
      izdnl1=0;bm=sdva[k];mb=bm-1;
      while(izdnl<a0){
         u0=dnl;
         while((u0>0)&&(a[izdnl1+u0]==0))u0--;
         u[i][0]=u0;up[i]=a[izdnl1+1] & mb;
         for(j=1;j<=u0;j++)u[i][j]=a[izdnl1+j];
         i++;izdnl1=izdnl;izdnl+=dnl;
      }
      u0=a0-izdnl1;u[i][0]=u0;up[i]=a[izdnl1+1] & mb;
      for(j=1;j<=u0;j++)u[i][j]=a[izdnl1+j];
      for(j=i+1;j<bm;j++){
         u[j][0]=up[j]=0;
      }
      mult(up,k);
      for(i=0;i<bm;i++)
         up[i]=((up[i] & mb)-(up[bm+i] & mb)+bm) & mb;
      for(i=1;i<bm;i++)pomeri(&u[i][0],i*sfi,l+1);
      fft(u,k,l+1,somg);
      for(i=0;i<bm;i++){
         kvadriraj(&u[i][0],l+1);
      }
      ffti1(u,k,l+1,-somg);
      for(i=0;i<bm;i++)pomeri(&u[i][0],-(i*sfi+k),l+1);
      for(i=0;i<bm;i++){
         if(u[i][0]>0)up[i]=(up[i]-(u[i][1]&mb)+bm)&mb;
      }
      for(i=0;i<bm;i++)sracunaj(up[i],i,k,l+1,&u[i][0]);
      a[0]=0;b[0]=0;
      for(i=0;i<bm;i++){
         ip=i*sdva[l-SB];
```

```
            if(u[i][0]<0){
               u[i][0]*=(-1);
               dodaj(b,&u[i][0],ip);
            }
            else{
               dodaj(a,&u[i][0],ip);
            }
         }
         moduo(a,n);moduo(b,n);
         modsub(a,b,a,n);
      }
   }/* kvadriraj */

   /* SQUARE: Kvadrira dati broj po algoritmu Karacuba & Hofman */
   void square(int *p){
      int ap,tp,k,i,u[DUZN],v[DUZN],z[DUZN],u0,z0,v0,p0;
      ap=p[0];
      if(ap<0)ap=(-ap);
      if(ap<5){
         switch(p[0]){
            case 0:
            case 1: p[0]=0;
                  if(ap==1){
                     tp=p[1]*p[1];
                     p[1]=tp & (BAZA-1);
                     p[2]=tp>>DZB;
                     if(p[2])p[0]=2;
                     else p[0]=1;
                  }
                  break;
            case 2: p[3]=0;
            case 3: p[4]=0;
            case 4: u[0]=p[1];u[1]=p[2];u[2]=u[0]-u[1];
                  u[3]=p[3];u[4]=p[4];u[5]=u[3]-u[4];
                  u[7]=p[2]-p[4];
                  if(u[7]==0){
                     u[8]=u[6]=p[1]-p[3];
                  }
                  else{
                     if(u[7]<0){
                        u[6]=p[3]-p[1];u[7]=-u[7];
                     }
                     else u[6]=p[1]-p[3];
                     if(u[6]<0){
                        u[6]+=BAZA;u[7]--;
                     }
                     u[8]=u[6]-u[7];
                  }
                  for(i=0;i<=8;i++){
                     tp=u[i]*u[i];
                     v[2*i]=tp&(BAZA-1);v[2*i+1]=tp>>DZB;
                  }
                  z[1]=v[0]+v[2]-v[4];z[3]=v[0]+v[6]-v[12];
                  z[5]=v[2]+v[8]-v[14];z[7]=v[6]+v[8]-v[10];
                  z[4]=z[1]+z[7]-(v[12]+v[14]-v[16]);
                  u[1]=v[1]+v[3]-v[5];u[3]=v[1]+v[7]-v[13];
                  u[5]=v[3]+v[9]-v[15];u[7]=v[7]+v[9]-v[11];
                  u[4]=u[1]+u[7]-(v[13]+v[15]-v[17]);
                  p[1]=v[0];p[2]=v[1]+z[1];p[3]=u[1]+v[2]+z[3];
                  p[4]=v[3]+u[3]+z[4];p[5]=u[4]+z[5]+v[6];
                  p[6]=u[5]+v[7]+z[7];p[7]=u[7]+v[8];p[8]=v[9];
                  p[9]=0;
                  for(i=1;i<=8;i++){
```

```
            while(p[i]<0){
               p[i]+=BAZA;p[i+1]--;
            }
            while(p[i]>=BAZA){
               p[i]-=BAZA;p[i+1]++;
            }
         }
         p0=9;
         while((p0>0)&&(p[p0]==0))p0--;
         p[0]=p0;
         break;
   }
}
else{
   if(ap%2){
      ap++;p[ap]=0;
   }
   k=ap/2;
   for(i=1;i<=k;i++){
      u[i]=p[i];v[i]=p[k+i];z[i]=u[i]-v[i];
   }
   u0=k;v0=k;
   while((u0>0) && (u[u0]==0))u0--;
   while((v0>0) && (v[v0]==0))v0--;
   u[0]=u0;v[0]=v0;z0=k;
   while((z0>0) && (z[z0]==0))z0--;
   if(z0){
      if(z[z0]<0)
        for(i=1;i<=z0;i++) z[i]= -z[i];
      for(i=1;i<=z0;i++){
         if(z[i]<0){
            z[i]+=BAZA;
            z[i+1]-=1;
         }
      }
      while((z0>0)&&(z[z0]==0))z0--;z[0]=z0;
      square(z);
   }
   else
      z[0]=0;
   square(u);square(v);
   u0=u[0];v0=v[0];
   for(i=1;i<=u0;i++)p[i]=u[i];
   for(i=1;i<=v0;i++)p[i+2*k]=v[i];
   for(i=u0+1;i<=2*k;i++)p[i]=0;
   p0=p[0]=2*k+v0;
   add(u,v);
   sub(u,z);
   u0=u[0];p[p0+1]=0;
   for(i=1;i<=u0;i++){
      p[i+k]+=u[i];
      if(p[i+k]>=BAZA){
         p[i+k]-=BAZA;
         p[i+k+1]+=1;
      }
   }
   i=u0+k+1;
   while(p[i]>=BAZA){
      p[i]-=BAZA;i++;
      p[i]+=1;
   }
   if(p[p0+1])p[0]+=1;
}
```

```
} /*square */

void add(int *a,int *b){ /* Sabira date brojeve */
   int a0,min,max,i;
   a0=a[0];min=b[0];
   if(a0<min){
      for(i=a0+1;i<=min;i++)a[i]=b[i];
      max=min;min=a0;
   }
   else max=a0;
   a[max+1]=0;
   for(i=1;i<=min;i++){
      a[i]+=b[i];
      if(a[i]>=BAZA){
         a[i]-=BAZA;
         a[i+1]+=1;
      }
   }
   i=min+1;
   while(a[i]>=BAZA){
      a[i]-=BAZA;i++;
      a[i]+=1;
   }
   if(a[max+1])
      a[0]=max+1;
   else
      a[0]=max;
   } /* add */

void sub(int *a,int *b){ /* Nalazi (a-b), pretpostavlja a>b */
   int a0,b0,i;
   b0=b[0];a0=a[0];
   for(i=1;i<=b0;i++){
      a[i]-=b[i];
      if(a[i]<0){
         a[i]+=BAZA;
         a[i+1]-=1;
      }
   }
   i=b0+1;
   while(a[i]<0){
      a[i]+=BAZA;i++;
      a[i]-=1;
   }
   while((a0>0)&&(a[a0]==0))a0--;
   a[0]=a0;
} /* sub */

void moduo(int *a,int m){   /* Nalazi (a mod F_m)  */
   int nv,a0,i;
   nv=sdva[m-SB];
   if(a[0]>nv){
      a0=a[0]-nv;a[1]-=a[nv+1];a[nv+1]=0;a[0]=nv;
      if(a[1]<0){
         a[1]+=BAZA;a[2]-=1;
      }
      for(i=2;i<=a0;i++){
         a[i]-=a[nv+i];
         if(a[i]<0){
            a[i]+=BAZA;
            a[i+1]-=1;
         }
      }
```

```
    i=a0+1;
    while((i<=nv)&&(a[i]<0)){
      a[i]+=BAZA;i++;
      a[i]-=1;
    }
    if(a[nv+1]<0){
      a[1]+=1;
      i=1;
      while(a[i]==BAZA){
        a[i]=0;i++;
        a[i]+=1;
      }
      if(a[nv+1]==0){
        a[0]=(-1);a[1]=1;
      }
    }
    a0=a[0];
    while((a0>0)&&(a[a0]==0))a0--;
    a[0]=a0;
  }
}/* moduo */



/* UN3.C */
#include <stdio.h>
#include <conc.h>
#include "uzag.h"
extern int sdva[21],t[TKX][TKY];

/* POMERI:
      Vrsi pomeranje datog broja ( b ) za pom mesta ulevo.
      Tj. mnozenje sa 2^pom . Sve to po modulu F_m.
*/
void pomeri(int *b,int pom,int m){
  int nv,sq,pom1,b0,mask,i,bz1;
  if((b[0]!=0) && (pom!=0)){
    nv=sdva[m];sq=1;
    while(pom<0)pom+=(2*nv);
    while(pom>=(2*nv))pom-=(2*nv);
    if(pom>=nv){
      pom-=nv;sq=-1;
    }
    pom1=pom%DZB;
    pom/=DZB;b0=b[0];
    if(b0==-1){
      if(sq<0){
        for(i=1;i<=pom;i++)b[i]=0;
        b[pom+1]=sdva[pom1];b[0]=pom+1;
      }
      else{
        if((DZB*pom+pom1)!=0){
          b[1]=0;nv=nv/DZB;
          for(i=2;i<=pom;i++)b[i]=0;
          b[pom+1]=BAZA-sdva[pom1];
          bz1=BAZA-1;
          for(i=pom+2;i<=nv;i++)b[i]=bz1;
          b[0]=nv;b[1]+=1;
        }
      }
    }
    else{
      if(pom1>0){
```

```
            b0++;b[b0]=0;mask=sdva[DZB-pom1]-1;
            for(i=b0;i>1;i--){
               b[i]=((b[i]&mask)<<pom1)+(b[i-1]>>(DZB-pom1));
            }
            b[1]=(b[1]&mask)<<pom1;
            if(b[b0]==0)b0--;
         }
         if(pom!=0){
            for(i=b0;i>0;i--)b[i+pom]=b[i];
            for(i=1;i<=pom;i++)b[i]=0;
         }
         b[0]=b0+pom;
         moduo(b,m);
         if(sq<0){
            pretvori(b,m);
         }
      }
   }
}/* pomeri */

/* MODSUB:
      Nalazi razliku datih brojeva po modulu F_m.
*/
void modsub(int *a,int *b,int *c,int m){
   int nv,i,j,a0,min,max,c0,bz1,b0;
   nv=sdva[m-SB];
   if((a[0]==-1)||(b[0]==-1)){
      if(a[0]==-1){
         if(b[0]==-1){
            c[0]=0;c[1]=0;
         }
         else{
            if(b[0]==0){
               c[0]=-1;c[1]=1;
            }
            else{
               i=1;b0=b[0];
               while((i<b0)&&(b[i]==0)){
                  c[i]=0;i++;
               }
               c[i]=BAZA-b[i];bz1=BAZA-1;
               for(j=i+1;j<=b0;j++)
                  c[j]=bz1-b[j];
               for(j=b0+1;j<=nv;j++)
                  c[j]=bz1;
               c0=nv;
               while((c0>0)&&(c[c0]==0))c0--;c[0]=c0;
            }
         }
      }
      else{
         if(a[0]==0){
            c[0]=c[1]=1;
         }
         else{
            c[1]=a[1]+1;i=1;c[nv+1]=c[a0+1]=a[a0+1]=0;
            while(c[i]>=BAZA){
               c[i]-=BAZA;i++;
               c[i]=a[i]+1;
            }
            for(j=i+1;j<=a0+1;j++)c[j]=a[j];
            if(c[nv+1]>0){
               c[0]=-1;c[1]=1;
```

```
                }
                else{
                  if(c[a0+1]>0){
                    c[0]=a0+1;
                  }
                  else c[0]=a0;
                }
              }
            }
          }
          else{
            a0=a[0];min=b[0];
            if(a0<min){
              max=min;min=a0;
            }
            else max=a0;
            for(i=1;i<=min;i++)c[i]=a[i]-b[i];
            if(a0==max){
              for(i=min+1;i<=max;i++)c[i]=a[i];
            }
            else{
              for(i=min+1;i<=max;i++)c[i]=-b[i];
            }
            c0=max;c[c0+1]=0;
            while((c0>0)&&(c[c0]==0))c0--;
            if(c0==0)c[0]=0;
            if(c[c0]<0){
              if((c0==1)&&(c[1]==-1)){
                c[0]=-1;c[1]=1;
              }
              else{
                c[1]+=1;i=1;
                while(c[i]==BAZA){
                  c[i]=0;i++;
                  c[i]+=1;
                }
                for(j=i;j<c0;j++){
                  if(c[j]<0){
                    c[j]+=BAZA;c[j+1]-=1;
                  }
                }
                c[c0]+=BAZA;bz1=BAZA-1;
                for(i=c0+1;i<=nv;i++)c[i]=bz1;
                c0=nv;
                while((c0>0)&&(c[c0]==0))c0--;
                c[0]=c0;
              }
            }
            else{
              for(i=1;i<=c0;i++){
                if(c[i]<0){
                  c[i]+=BAZA;c[i+1]-=1;
                }
              }
              while((c0>0)&&(c[c0]==0))c0--;
              c[0]=c0;
            }
          }
        } /* modsub */

        /* PRETVORI:
             Datom negativnom broju dodaje F_m, kako bi dobio
             najmanju pozetivnu vrednost po modulu F_m.
```

```
*/
void pretvori(int *b,int m){
   int nv,i,b0,bz1,j;
   if((b[0]==-1)||((b[0]==1)&&(b[1]==1)))b[0]*=(-1);
   else{
      if(b[0]!=0){
         nv=sdva[m-SB];b[1]=1-b[1];
         i=2;b0=b[0];bz1=BAZA-1;
         if(b[1]>=0){
            while(b[i]==0)i++;
            b[i]=BAZA-b[i];i++;
         }
         else{
            b[1]+=BAZA;
         }
         for(j=i;j<=b0;j++)b[j]=bz1-b[j];
         for(j=b0+1;j<=nv;j++)b[j]=bz1;
         b0=nv;
         while((b0>0)&&(b[b0]==0))b0--;b[0]=b0;
      }
   }
}/* pretvori */

/* FFT: Nalazi FT datog niza.  */
void fft(int x[][DUZ_UJ1],int k,int m,int somg){
   int j,sdvkj1,sdvj,sdvj1,s,szdj1,i,p1,p2,q1,xp0,q,tx[2*DUZ_UJ1];
   for(j=0;j<k;j++){
      sdvkj1=sdva[k-j-1];sdvj=sdva[j];sdvj1=2*sdvj;
      for(s=0;s<sdvkj1;s++){
         szdj1=s*sdvj1;
         for(i=0;i<sdvj;i++){
            p1=t[k][szdj1+i];
            p2=t[k][szdj1+i+sdvj];
            xp0=x[p2][0];
            for(q=0;q<=xp0;q++)tx[q]=x[p2][q];
            pomeri(tx,somg*i*sdvkj1,m);
            modsub(&x[p1][0],tx,&x[p2][0],m);
            if((x[p1][0]==-1)||(tx[0]==-1)){
               if(tx[0]==-1){
                  tx[0]=1;
                  modsub(&x[p1][0],tx,&x[p1][0],m);
               }
               else{
                  if(tx[0]>0){
                     xp0=tx[0];
                     x[p1][1]=tx[1]-1;q=1;
                     while(x[p1][q]<0){
                        x[p1][q]+=BAZA;q++;
                        x[p1][q]=tx[q]-1;
                     }
                     for(q1=q+1;q1<=xp0;q1++)
                       x[p1][q1]=tx[q1];
                     while((xp0>0)&&(x[p1][xp0]==0))xp0--;
                     x[p1][0]=xp0;
                  }
               }
            }
            else{
               add(&x[p1][0],tx);
               modulo(&x[p1][0],m);
            }
         }
      }
```

```
    }
}/* fft */

/* FFTI: Nalazi inverzne FT od datog niza. */
void ffti(int x[][DUZ_UJ],int k,int m,int somg){
   int j,sdvkj1,sdvj,sdvj1,s,szdj1,i,p1,p2,xp0,q,q1,tx[2*DUZ_UJ];
   for(j=0;j<k;j++){
      sdvkj1=sdva[k-j-1];sdvj=sdva[j];sdvj1=2*sdvj;
      for(s=0;s<sdvkj1;s++){
         szdj1=s*sdvj1;
         for(i=0;i<sdvj;i++){
            p1=szdj1+i;
            p2=p1+sdvj;
            xp0=x[p2][0];
            for(q=0;q<=xp0;q++)tx[q]=x[p2][q];
            pomeri(tx,somg*i*sdvkj1,m);
            modsub(&x[p1][0],tx,&x[p2][0],m);
            if((x[p1][0]==-1)||(tx[0]==-1)){
               if(tx[0]==-1){
                  tx[0]=1;
                  modsub(&x[p1][0],tx,&x[p1][0],m);
               }
               else{
                  if(tx[0]>0){
                     xp0=tx[0];
                     x[p1][1]=tx[1]-1;q=1;
                     while(x[p1][q]<0){
                        x[p1][q]+=BAZA;q++;
                        x[p1][q]=tx[q]-1;
                     }
                     for(q1=q+1;q1<=xp0;q1++)
                        x[p1][q1]=tx[q1];
                     while((xp0>0)&&(x[p1][xp0]==0))xp0--;
                     x[p1][0]=xp0;
                  }
               }
            }
            else{
               add(&x[p1][0],tx);
               moduo(&x[p1][0],m);
            }
         }
      }
   }
}/* ffti */

/* FFTI1: Isto kao  FFTI samo sa manjim dimenzijama niza u... */
void ffti1(int x[][DUZ_UJ1],int k,int m,int somg){
   int j,sdvkj1,sdvj,sdvj1,s,szdj1,i,p1,p2,xp0,q,q1,tx[2*DUZ_UJ1];
   for(j=0;j<k;j++){
      sdvkj1=sdva[k-j-1];sdvj=sdva[j];sdvj1=2*sdvj;
      for(s=0;s<sdvkj1;s++){
         szdj1=s*sdvj1;
         for(i=0;i<sdvj;i++){
            p1=szdj1+i;
            p2=p1+sdvj;
            xp0=x[p2][0[;
            for(q=0;q<=xp0;q++)tx[q]=x[p2][q];
            pomeri(tx,somg*i*sdvkj1,m);
            modsub(&x[p1][0],tx,&x[p2][0],m);
            if((x[p1][0]==-1)||(tx[0]==-1)){
               if(tx[0]==-1){
                  tx[0]=1;
```

```
                modsub(&x[p1][0],tx,&x[p1][0],m);
              }
              else{
                if(tx[0]>0){
                  xp0=tx[0];
                  x[p1][1]=tx[1]-1;q=1;
                  while(x[p1][q]<0){
                    x[p1][q]+=BAZA;q++;
                    x[p1][q]=tx[q]-1;
                  }
                  for(q1=q+1;q1<=xp0;q1++)
                    x[p1][q1]=tx[q1];
                  while((xp0>0)&&(x[p1][xp0]==0))xp0--;
                  x[p1][0]=xp0;
                }
              }
            }
            else{
              add(&x[p1][0],tx);
              moduo(&x[p1][0],m);
            }
          }
        }
      }
    }/* ffti */


    /* UN4.C */
    #define BAZA 65536  /* vrednost baze */
    #define SB 4  /* stepen baze */
    extern int sdva[21];
    extern int stri[9];

    /* MULT:
          Izvrsava cetvrti korak Schonhage Strassenovog algoritma.
          Tj. nalazi konvoluciju niza y[] sa samim sobom po datom
          modulu. Zasniva se na Karatsuba & Ofmanovom algoritmu za
          mnozenje polinoma. Ovde je implementirana nerekurzivna
          verzija.
    */
    void mult(int *y,int k){
      int s,s1,s2,p,i,j,l,t,r,r1,r2,r3,r4,q,q1,q2,q3,q4,q5,q6;
      static int a[2][1460],b[10];
      if(k>1){
        for(i=0;i<sdva[k];i++)a[0][i]=y[i];
        s1=0;s2=1;
        for(j=k;j>2;j--){
          for(p=0;p<stri[k-j];p++){
            q=p*sdva[j];r=3*p*sdva[j-1];
            for(l=0;l<sdva[j-1];l++){
              a[s2][r+l]=a[s1][q+l];
              a[s2][r+l+sdva[j-1]]=a[s1][q+l+sdva[j-1]];
              a[s2][r+l+sdva[j]]=a[s2][r+l]-a[s2][r+l+sdva[j-1]];
            }
          }
          t=s1;s1=s2;s2=t;
        }
        for(p=0;p<stri[k-2];p++){
          q=4*p;r=8*p;
          b[0]=a[s1][q];b[1]=a[s1][q+1];b[2]=b[0]-b[1];
          b[3]=a[s1][q+2];b[4]=a[s1][q+3];b[5]=b[3]-b[4];
          b[6]=b[0]-b[3];b[7]=b[1]-b[4];b[8]=b[6]-b[7];
```

```
            for(i=0;i<9;i++)b[i]=b[i]*b[i];
            a[s2][r]=b[0];a[s2][r+1]=b[0]+b[1]-b[2];
            a[s2][r+2]=b[0]+b[1]+b[3]-b[6];
            a[s2][r+4]=b[1]+b[4]-b[7]+b[3];
            a[s2][r+5]=b[3]+b[4]-b[5];
            a[s2][r+6]=b[4];a[s2][r+7]=0;
            a[s2][r+3]=a[s2][r+1]+a[s2][r+5]-(b[6]+b[7]-b[8]);
          }
          t=s1;s1=s2;s2=t;
          for(s=2;s<k;s++){
            for(l=0;l<stri[k-s-1];l++){
              q=3*l*sdva[s+1];r=l*sdva[s+2];
              for(i=0;i<sdva[s];i++){
                r1=r+i;r2=r1+sdva[s];r3=r2+sdva[s];r4=r3+sdva[s];
                q1=q+i;q2=q1+sdva[s];q3=q2+sdva[s];q4=q3+sdva[s];
                q5=q4+sdva[s];q6=q5+sdva[s];
                a[s2][r1]=a[s1][q1];a[s2][r4]=a[s1][q4];
                a[s2][r2]=a[s1][q1]+a[s1][q2]+a[s1][q3]-a[s1][q5];
                a[s2][r3]=a[s1][q2]+a[s1][q4]+a[s1][q3]-a[s1][q6];
              }
            }
            t=s1;s1=s2;s2=t;
          }
          for(i=0;i<sdva[k+1];i++)y[i]=a[s1][i];
        }
        else{
          if(k>0){
            y[3]=0;y[2]=y[0]-y[1];
            y[0]*=y[0];y[1]*=y[1];y[2]*=y[2];
          }
          else{
            y[0]=y[0]*y[0];
            y[1]=0;
          }
        }
      }/* mult */

      void dodaj(int *a,int *y,int ip){ /* Broju a dodaje broj  y*(BAZA)^(ip) */
        int a0,y0,i,tmp;
        a0=a[0];y0=y[0];
       if(y0!=0){
        if(a0<ip){
          for(i=a0+1;i<=ip;i++)a[i]=0;
          for(i=1;i<=y0;i++)a[ip+i]=y[i];
          a[0]=ip+y0;
        }
        else{
          if(a0<(ip+y0)){
            for(i=a0+1;i<=(ip+y0);i++)a[i]=y[i-ip];
            tmp=a0;a0=ip+y0;y0=tmp-ip;
          }
          a[a0+1]=0;
          for(i=1;i<=y0;i++){
            a[ip+i]+=y[i];
            if(a[ip+i]>=BAZA){
              a[ip+i]-=BAZA;
              a[ip+i+1]+=1;
            }
          }
          i=ip+y0+1;
          while(a[i]>=BAZA){
            a[i]-=BAZA;i++;
            a[i]+=1;
```

```
        }
        if(a[a0+1]!=0)a0++;
        a[0]=a0;
      }
    }
  } /* dodaj */

  /* SRACUNAJ: Ostvaruje peti korak Schonhage Strassenovog algoritma. */
  void sracunaj(int yp,int ip,int k,int l,int *y){
    int nv,i,bz1,y0,j,spc;
    if(y[0]==-1){
      nv=sdva[l-SB];
      if(ip>yp){
        y[1]=yp;y[nv+1]=yp+1;y[0]=nv+1;
        for(i=2;i<=nv;i++)y[i]=0;
      }
      else{
        yp=sdva[k]-yp;y[1]=yp;
        if(yp==1)y[0]=-1;
        else{
          y[nv+1]=yp-1;y[0]=-(nv+1);
          for(i=2;i<=nv;i++)y[i]=0;
        }
      }
    }
    else{
      bz1=BAZA-1;
      if(yp!=0){
        nv=sdva[l-SB];
        if(yp>ip){
          y0=y[0];yp=sdva[k]-yp;
          if(y0==0){
            y[1]=y[nv+1]=yp;y[0]=-(nv+1);
            for(i=2;i<=nv;i++)y[i]=0;
          }
          else{
            y[1]=yp-y[1];
            if(y[1]<0){
              y[1]+=BAZA;
              for(i=2;i<=y0;i++)y[i]=bz1-y[i];
              for(i=y0+1;i<=nv;i++)y[i]=bz1;
              y[nv+1]=yp-1;
              if(yp==1)y[0]=-nv;
              else y[0]=-(nv+1);
            }
            else{
              if(y0>1){
                i=2;
                while(y[i]==0)i++;
                y[i]=BAZA-y[i];
                for(j=i+1;j<=y0;j++)y[j]=bz1-y[j];
                for(i=y0+1;i<=nv;i++)y[i]=bz1;
                y[nv+1]=yp-1;
                if(yp==1)y[0]=-nv;
                else y[0]=-(nv+1);
              }
              else{
                for(i=2;i<=nv;i++)y[i]=0;
                y[nv+1]=yp;y[0]=-(nv+1);
              }
            }
          }
        }/* if(yp>ip) */
```

```
        else{
          spc=0;y0=y[0];
          if(y0==0){
            y[1]=y[nv+1]=yp;
            for(i=2;i<=nv;i++)y[i]=0;
            y[0]=nv+1;
          }
          else{
            if((yp==ip)&&(y0==nv)&&((y[1]+ip)>=BAZA)){
              i=nv;
              while(y[i]==bz1)i--;
              if(i<2){
                spc=1;yp=sdva[k]-yp;y[nv+1]=yp-1;
                if(yp==1){
                  y[1]=yp-y[1];
                  if(y[1]<0){
                    y[1]+=BAZA;y[0]=-1;
                  }
                  else{
                    y[2]=1;y[0]=-2;
                  }
                }
                else{
                  for(i=3;i<=nv;i++)y[i]=0;
                  y[0]=-(nv+1);y[1]=yp-y[1];
                  if(y[1]<0){
                    y[1]+=BAZA;y[2]=0;
                  }
                  else{
                    y[2]=1;
                  }
                }
              }
            }/* if((yp==ip)&&... */
            if(spc==0){
              y[nv+1]=y[y0+1]=0;
              y[1]+=yp;i=1;
              while(y[i]>=BAZA){
                y[i]-=BAZA;i++;
                y[i]+=1;
              }
              for(i=y0+2;i<=nv;i++)y[i]=0;
              y[nv+1]+=yp;
              y[0]=nv+1;
            }
          }
        }
      }
    }
  } /* sracunaj */



  /* UNM.C */
  #include <stdio.h>
  #include <conc.h>
  #include "uzag.h"
  #include "zph.h"
  int sdva[21],t[TKX][TKY];
  /* MAIN: Ostvaruje Pepinov test pozivajuci proceduru pkvadriraj,
        za paralelno kvadriranje. Dobijenu vrednost tj.
        3^{2^(2^n - 1)} (mod F_n), ce biti sacuvana u datoteci
```

```
        "fedre.dat". Povremeno se pamte i medjurezultati, tako
        da u slucaju prekida programa nije potrebno ponavljati
        citav racun.
*/
main(){
  static int a[2*DUZ_A];
  int i,j,n,dd;
  FILE *fp,*fp1;
  printf("\n Unesite n(>5): ");scanf("%d",&n);getchar();
  printf("\n Zelite li da nastavite unesite !=0,a za pocetak unesite 0: ");
  scanf("%d",&j);getchar();
  if(j==0){
   a[0]=1;a[1]=3;dd=1;
  }
  else{
    printf("\n Za nastavak od prve datoteke unesite 1,a od druge 2: ");
    scanf("%d",&j);getchar();
    if(j==2){
      fp1=fopen("femr2.dat","r+");
    }
    else{
      fp1=fopen("femr1.dat","r+");
    }
    uzmi(a,fp1,&dd);
    fclose(fp1);
  }
  inic();
  printf("\n Poco sam n=%d, dd=%d, a[0]=%d \n",n,dd,a[0]);
  SetTime(-1000000000);
  for(i=dd;i<sdva[n];i++){
    pkvadriraj(a,n,i);
  }
  printf("\n Vreme= %d ",Time());
  fp=fopen("fedre.dat","w+");
  smesti(a,fp,0);
  printf("\n Kraj %d \n",a[0]);
  ChanOutInt(P0O,0);
}/* main */

/* SMESTI: Upisuje u odgovarajucu datoteku dati broj. */
void smesti(int *a,FILE *fp,int dd){
  int i,j;
  i=0;
  fprintf(fp," %d %d\n",a[0],dd);
  while(i<a[0]){
    j=0;
    while((j<12) && (i<a[0])){
      j++;i++;
      fprintf(fp," %x",a[i]);
    }
    fprintf(fp,"\n");
  }
}/* smesti */

/* UZMI:
      Ucitava veliki broj iz datoteke i smesta u niz.
      Pretpostavka je da se u jednoj reci pamti osmobitni broj.
*/
void uzmi(int *a,FILE *fp,int *dd){
  int i,j,t;
  i=0;
  fscanf(fp," %d %d\n",&a[0],&t);
  *dd=t;
```

```
        while(i<a[0]){
          j=0;
          while((j<12) && (i<a[0])){
            j++;i++;
            fscanf(fp," %x",&a[i]);
          }
          fscanf(fp,"\n");
        }
      }/* uzmi */

      /* PKVADRIRAJ: Vrsi kvadriranje datog broja po modulu F_n. Procedura
                 se zasniva na paralelnom Schonhage Strassenovom algoritmu.
      */
      void pkvadriraj(int *a,int n,int ddd){
        int l,k,a0,i,dnl,bm,mb,j,nbm;
        static int up[2049];
        FILE *fd1,*fd2;
        if(a[0]<33){
          kvadriraj(a,n);
        }
        else{
          while(a[0]<=sdva[n-2-SB])n--;
          ChanOutInt(P0O,n);sendu(a,P0O);
          if((ddd%3000)==0){
            printf("\n %d %x NE gasi me radim ",ddd,a[1]);
            if((ddd%6000)==0){
              fd1=fopen("femr2.dat","w+");
              smesti(a,fd1,ddd);
              fclose(fd1);
            }
            else{
              fd2=fopen("femr1.dat","w+");
              smesti(a,fd2,ddd);
              fclose(fd2);
            }
          }
          l=n/2;k=n-l;
          nbm=sdva[k-SBP];
          a0=a[0];i=0;dnl=sdva[l-SB];
          bm=sdva[k];mb=bm-1;
          while((i*dnl+1)<=a0){
            up[i]=a[i*dnl+1] & mb;i++;
          }
          for(j=i;j<bm;j++)up[j]=0;
          mult1(up,k);
          for(i=0;i<bm;i++)
            up[i]=((up[i] & mb)-(up[bm+i] & mb)+bm) & mb;
          for(i=0;i<bm;i++)ChanOutInt(P0O,up[i]);
          receiveu(a,P0I);
        }
      }/* pkvadriraj */

      /* MULT1:
            Izvrsava cetvrti korak Schonhage Strassenovog algoritma.
            Tj. nalazi konvoluciju niza y[] sa samim sobom po datom
            modulu. Zasniva se na Karatsuba & Ofmanovom algoritmu za
            mnozenje polinoma. Procedura je rekurzivna, kao izlaz iz
            rekurzije koristi nerekurzivnu varijantu ove procedure
            MULT(...).
      */
      void mult1(int *y,int k){
        int s,i,dvas,tris,a[1025],b[1025],c[1025];
        if(k>6){
```

```
        s=sdva[k-1];
        for(i=0;i<s;i++){
           a[i]=y[i];b[i]=y[i+s];c[i]=a[i]-b[i];
        }
        mult1(a,k-1);mult1(b,k-1);mult1(c,k-1);
        dvas=2*s;tris=3*s;
        for(i=0;i<s;i++){
           y[i]=a[i];y[i+tris]=b[i+s];
           y[i+s]=a[i+s]+a[i]+b[i]-c[i];
           y[i+dvas]=b[i]+a[i+s]+b[i+s]-c[i+s];
        }
     }
     else{
        mult(y,k);
     }
  } /* mult */



  /* UP0.C :
        Program koji izvrsava procesor P0
  */
  #include <stdio.h>
  #include <conc.h>
  #include "uzag.h"
  #include "zp0.h"
  int sdva[21],t[TKX][TKY];
  main(){
     int n,i,j,l,k,sfi,somg,a0,m,dnl,bm,mb,nbm,ip,it,x0;
     static int a[2*DUZ_A],b[2*DUZ_A],a1[2*DUZ_A1],b1[2*DUZ_A1];
     static int x[DUZ_UI][DUZ_UJ],y[DUZ_UI][DUZ_UJ];
     inic();
     n=ChanInInt(PHI);
     while(n!=0){ /* kada primi 0 onda se zaustavlja proces kvadriranja */
        /* razmena podataka */
        ChanOutInt(P1O,n);ChanOutInt(P2O,n);
        ChanOutInt(P4O,n);receiveu(a,PHI);
        l=n/2;k=n-l;
        sfi=sdva[l+1-k];somg=2*sfi;
        a0=a[0];dnl=sdva[l-SB];m=l+1;
        bm=sdva[k];mb=bm-1;
        ChanOutInt(P1O,a0);
        predaj(1,a0,dnl,2,a,P1O);
        ChanOutInt(P2O,a[0]);
        predaj(2,a[0],dnl,4,a,P2O);
        ChanOutInt(P4O,a[0]);
        predaj(4,a[0],dnl,8,a,P4O);
        pp(k,8,dnl,a[0],0,sfi,m,a,x);/* pripremna faza tj. FT nad
                                odgovarajucim podacima    */
        nbm=sdva[k-SBP];
        /* razmena podataka sa procesorom P4 ... u cilju dobijanja
          FT vecih podnizova    */
        for(i=0;i<nbm/8;i++){
           sendu(&x[8*i+4][0],P4O);sendu(&x[8*i+5][0],P4O);
           sendu(&x[8*i+6][0],P4O);sendu(&x[8*i+7][0],P4O);
        }
        for(i=0;i<nbm/8;i++){
           receiveu(&y[4*i][0],P4I);receiveu(&y[4*i+1][0],P4I);
           receiveu(&y[4*i+2][0],P4I);receiveu(&y[4*i+3][0],P4I);
        }
        for(i=0;i<nbm/8;i++){
           abc2(&x[8*i][0],&y[4*i][0],&y[4*i+nbm/2][0],somg*4*(8*i),m);
           abc2(&x[8*i+1][0],&y[4*i+1][0],&y[4*i+1+nbm/2][0],somg*4*(8*i+1),m);
```

```
        abc2(&x[8*i+2][0],&y[4*i+2][0],&y[4*i+2+nbm/2][0],somg*4*(8*i+2),m);
        abc2(&x[8*i+3][0],&y[4*i+3][0],&y[4*i+3+nbm/2][0],somg*4*(8*i+3),m);
      }
      /* razmena podataka sa procesorom P2 ... u cilju dobijanja
        FT vecih podnizova    */
      for(i=0;i<nbm/4;i++){
        sendu(&y[4*i+2][0],P2O);sendu(&y[4*i+3][0],P2O);
      }
      for(i=0;i<nbm/4;i++){
        receiveu(&x[2*i][0],P2I);receiveu(&x[2*i+1][0],P2I);

      }
      for(i=0;i<nbm/4;i++){
       abc2(&y[4*i][0],&x[2*i][0],&x[2*i+nbm/2][0],somg*2*(8*i),m);
       abc2(&y[4*i+1][0],&x[2*i+1][0],&x[2*i+1+nbm/2][0],somg*2*(8*i+1),m);
      }
      /* razmena podataka sa procesorom P1 ... u cilju dobijanja
        FT vecih podnizova    */
      for(i=0;i<nbm/2;i++)sendu(&x[2*i+1][0],P1O);
      for(i=0;i<nbm/2;i++)receiveu(&y[i][0],P1I);
      for(i=0;i<nbm/2;i++){
        abc2(&x[2*i][0],&y[i][0],&y[i+nbm/2][0],somg*(8*i),m);
      }
      for(i=0;i<nbm;i++){
        it=t[k-SBP][i];x0=y[it][0];
        for(j=0;j<=x0;j++)x[i][j]=y[it[j];
      }
      /* drugi korak ... tj. kvadriranje odgovarajucih komponenti */
      for(i=0;i<nbm;i++){
        kvadriraj(&x[i][0],m);
      }
      /* pocetak inverznih transformacija */
      somg*=(-1);
      ffti(x,k-SBP,m,somg*8);
      for(i=nbm/2;i<nbm;i++)sendu(&x[i][0],P4O);
      for(i=0;i<nbm/2;i++)receiveu(&y[i][0],P4I);
      for(i=0;i<nbm/2;i++)abc1(&x[i][0],&y[i][0],&x[i+nbm/2][0],somg*4*i,m);
      for(i=nbm/2;i<nbm;i++)sendu(&x[i][0],P2O);
      for(i=0;i<nbm/2;i++)receiveu(&y[i][0],P2I);
      for(i=0;i<nbm/2;i++)abc1(&x[i][0],&y[i][0],&x[i+nbm/2][0],somg*2*i,m);
      for(i=nbm/2;i<nbm;i++)sendu(&x[i][0],P1O);
      for(i=0;i<nbm/2;i++)receiveu(&y[i][0],P1I);
      for(i=0;i<nbm/2;i++)abc1(&x[i][0],&y[i][0],&x[i+nbm/2][0],somg*i,m);
      for(i=0;i<nbm/2;i++){
        pomeri(&x[i][0],-(k+sfi*i),m);
        pomeri(&x[i+nbm/2][0],-(k+sfi*(i+bm/2)),m);
      }
      /* prosledjivanje podataka od procesora PH koji je izvrsio 4. korak */
      for(i=0;i<bm;i++){
        a1[i]=ChanInInt((PHI);
      }
      for(i=2*nbm;i<4*nbm;i++){
        ChanOutInt(P1O,a1[i]);
      }
      for(i=nbm;i<2*nbm;i++){
        ChanOutInt(P2O,a1[i]);
      }
      for(i=nbm/2;i<nbm;i++){
        ChanOutInt(P4O,a1[i]);
      }
      for(i=2*nbm+bm/2;i<(4*nbm+bm/2);i++){
        ChanOutInt(P1O,a1[i]);
      }
```

```
        for(i=nbm+bm/2;i<(2*nbm+bm/2);i++){
          ChanOutInt(P2O,a1[i]);
        }
        for(i=nbm/2+bm/2;i<(nbm+bm/2);i++){
          ChanOutInt(P4O,a1[i]);
        }
        for(i=0;i<nbm/2;i++){
          if(x[i][0]>0)a1[i]=(a1[i]-(x[i][1]&mb)+bm)&mb;
          if(x[i+nbm/2][0]>0){
            a1[i+nbm/2]=(a1[i+bm/2]-(x[i+nbm/2][1]&mb)+bm)&mb;
          }
          else a1[i+nbm/2]=a1[i+bm/2];
        }
        /* peti korak... */
        for(i=0;i<nbm/2;i++){
          sracunaj(a1[i],i,k,m,&x[i][0]);
          sracunaj(a1[i+nbm/2],i+bm/2,k,m,&x[i+nbm/2][0]);
        }
        /* sesti korak... tj. sumiranje */
        a1[0]=0;b1[0]=0;
        for(i=0;i<nbm/2;i++){
          ip=i*sdva[l-SB];
          if(x[nbm/2+i][0]<0){
            x[nbm/2+i][0]*=(-1);
            dodaj(b1,&x[nbm/2+i][0],ip);
          }
          else dodaj(a1,&x[nbm/2+i][0],ip);
        }
        sendu(a1,P4O);sendu(b1,P4O);
        receiveu(a1,P4I);receiveu(b1,P4I);
        a[0]=0;b[0]=0;
        for(i=0;i<nbm/2;i++){
          ip=i*sdva[l-SB];
          if(x[i][0]<0){
            x[i][0]*=(-1);
            dodaj(b,&x[i][0],ip);
          }
          else dodaj(a,&x[i][0],ip);
        }
        ip=(nbm/2)*sdva[l-SB];
        dodaj(a,a1,ip);dodaj(b,b1,ip);
        receiveu(a1,P2I);receiveu(b1,P2I);
        ip=nbm*sdva[l-SB];
        dodaj(a,a1,ip);dodaj(b,b1,ip);
        receiveu(a1,P1I);receiveu(b1,P1I);
        ip=2*nbm*sdva[l-SB];
        dodaj(a,a1,ip);dodaj(b,b1,ip);
        receiveu(a1,P4I);receiveu(b1,P4I);
        ip=4*nbm*sdva[l-SB];
        dodaj(a,a1,ip);dodaj(b,b1,ip);
        moduo(a,n);moduo(b,n);
        modsub(a,b,a,n);
        sendu(a,PHO); /* rezultat se predaje procesoru PH */
        n=ChanInInt(PHI); /* ceka pocetak novog kvadriranja... */
    }
    ChanOutInt(P1O,0);ChanOutInt(P2O,0);ChanOutInt(P4O,0);
}/* main */



/* UP1.C :
      Program koji izvrsava procesor P1
*/
```

```
#include <stdio.h>
#include <conc.h>
#include "uzag.h"
#include "zp1.h"
int sdva[21],t[TKX][TKY];
main(){
  int n,i,j,l,k,sfi,somg,a0,m,dnl,bm,mb,nbm,ip,it,x0;
  static int a[DUZ_A],b[DUZ_A],a1[DUZ_A1],b1[DUZ_A1];
  static int x[DUZ_UI][DUZ_UJ],y[DUZ_UI][DUZ_UJ];
  inic();
  n=ChanInInt(P0I);
  while(n!=0){ /* kada primi 0 onda se zaustavlja proces kvadriranja */
    /* razmena podataka */
    ChanOutInt(P3O,n);ChanOutInt(P5O,n);
    l=n/2;k=n-l;
    sfi=sdva[l+1-k];somg=2*sfi;
    dnl=sdva[l-SB];m=l+1;
    bm=sdva[k];mb=bm-1;
    a0=ChanInInt(P0I);
    primi(1,a0,dnl,2,a,P0I);
    ChanOutInt(P3O,a[0]);
    predaj(1,a[0],dnl,2,a,P3O);
    ChanOutInt(P5O,a[0]);
    predaj(2,a[0],dnl,4,a,P5O);
    pp(k,4,dnl,a[0],1,sfi,m,a,x);/* pripremna faza tj. FT nad
                          odgovarajucim podacima   */
    nbm=sdva[k-SBP];
    /* razmena podataka sa procesorom P5 ... u cilju dobijanja
      FT vecih podnizova   */
    for(i=0;i<nbm/8;i++){
      sendu(&x[8*i+4][0],P5O);sendu(&x[8*i+5][0],P5O);
      sendu(&x[8*i+6][0],P5O);sendu(&x[8*i+7][0],P5O);
    }
    for(i=0;i<nbm/8;i++){
      receiveu(&y[4*i][0],P5I);receiveu(&y[4*i+1][0],P5I);
      receiveu(&y[4*i+2][0],P5I);receiveu(&y[4*i+3][0],P5I);
    }
    for(i=0;i<nbm/8;i++){
     abc2(&x[8*i][0],&y[4*i][0],&y[4*i+nbm/2][0],somg*4*(8*i),m);
     abc2(&x[8*i+1][0],&y[4*i+1][0],&y[4*i+1+nbm/2][0],somg*4*(8*i+1),m);
     abc2(&x[8*i+2][0],&y[4*i+2][0],&y[4*i+2+nbm/2][0],somg*4*(8*i+2),m);
     abc2(&x[8*i+3][0],&y[4*i+3][0],&y[4*i+3+nbm/2][0],somg*4*(8*i+3),m);
    }
    /* razmena podataka sa procesorom P3 ... u cilju dobijanja
      FT vecih podnizova   */
    for(i=0;i<nbm/4;i++){
      sendu(&y[4*i+2][0],P3O);sendu(&y[4*i+3][0],P3O);
    }
    for(i=0;i<nbm/4;i++){
      receiveu(&x[2*i][0],P3I);receiveu(&x[2*i+1][0],P3I);
    }
    for(i=0;i<nbm/4;i++){
      abc2(&y[4*i][0],&x[2*i][0],&x[2*i+nbm/2][0],somg*2*(8*i),m);
      abc2(&y[4*i+1][0],&x[2*i+1][0],&x[2*i+1+nbm/2][0],somg*2*(8*i+1),m);
    }
    /* razmena podataka sa procesorom P0 ... u cilju dobijanja
      FT vecih podnizova   */
    for(i=0;i<nbm/2;i++)receiveu(&y[i][0],P0I);
    for(i=0;i<nbm/2;i++)sendu(&x[2*i][0],P0O);
    for(i=0;i<nbm/2;i++){
      abc1(&y[i][0],&x[2*i+1][0],&y[i+nbm/2][0],somg*(8*i+1),m);
    }
    for(i=0;i<nbm;i++){
```

```
        it=t[k-SBP][i];x0=y[it][0];
        for(j=0;j<=x0;j++)x[i][j]=y[it][j];
      }
      /* drugi korak ... tj. kvadriranje odgovarajucih komponenti */
      for(i=0;i<nbm;i++){
        kvadriraj(&x[i][0],m);
      }
      /* pocetak inverznih transformacija */
      somg*=(-1);
      ffti(x,k-SBP,m,somg*8);
      for(i=nbm/2;i<nbm;i++)sendu(&x[i][0],P5O);
      for(i=0;i<nbm/2;i++)receiveu(&y[i][0],P5I);
      for(i=0;i<nbm/2;i++)abc1(&x[i][0],&y[i][0],&x[i+nbm/2][0],somg*4*i,m);
      for(i=nbm/2;i<nbm;i++)sendu(&x[i][0],P3O);
      for(i=0;i<nbm/2;i++)receiveu(&y[i][0],P3I);
      for(i=0;i<nbm/2;i++)abc1(&x[i][0],&y[i][0],&x[i+nbm/2][0],somg*2*i,m);
      for(i=0;i<nbm/2;i++)receiveu(&y[i][0],P0I);
      for(i=0;i<nbm/2;i++)sendu(&x[i][0],P0O);
      for(i=0;i<nbm/2;i++)
        abc1(&y[i][0],&x[i+nbm/2][0],&y[i+nbm/2][0],somg*(i+2*nbm),m);
      for(i=0;i<nbm/2;i++){
        pomeri(&y[i][0],-(k+sfi*(i+2*nbm)),m);
        pomeri(&y[i+nbm/2][0],-(k+sfi*(i+2*nbm+bm/2)),m);
      }
      /* prosledjivanje podataka od procesora PH koji je izvrsio 4. korak */
      for(i=0;i<2*nbm;i++){
        a1[i]=ChanInInt(P0I);
      }
      for(i=nbm;i<2*nbm;i++){
        ChanOutInt(P3O,a1[i]);
      }
      for(i=nbm/2;i<nbm;i++){
        ChanOutInt(P5O,a1[i]);
      }
      for(i=2*nbm;i<4*nbm;i++){
        a1[i]=ChanInInt(P0I);
      }
      for(i=3*nbm;i<4*nbm;i++){
        ChanOutInt(P3O,a1[i]);
      }
      for(i=5*nbm/2;i<3*nbm;i++){
        ChanOutInt(P5O,a1[i]);
      }
      for(i=0;i<nbm/2;i++){
        if(y[i][0]>0)a1[i]=(a1[i]-(y[i][1]&mb)+bm)&mb;
        if(y[i+nbm/2][0]>0){
          a1[i+nbm/2]=(a1[i+2*nbm]-(y[i+nbm/2][1]&mb)+bm)&mb;
        }
        else a1[i+nbm/2]=a1[i+2*nbm];
      }
      /* peti korak... */
      for(i=0;i<nbm/2;i++){
        sracunaj(a1[i],i+2*nbm,k,m,&y[i][0]);
        sracunaj(a1[i+nbm/2],i+2*nbm+bm/2,k,m,&y[i+nbm/2][0]);
      }
      /* sesti korak... tj. sumiranje */
      a1[0]=0;b1[0]=0;
      for(i=0;i<nbm/2;i++){
        ip=i*sdva[l-SB];
        if(y[nbm/2+i][0]<0){
          y[nbm/2+i][0]*=(-1);
          dodaj(b1,&y[nbm/2+i][0],ip);
        }
```

```
            else dodaj(a1,&y[nbm/2+i][0],ip);
          }
          sendu(a1,P5O);sendu(b1,P5O);
          receiveu(a1,P5I);receiveu(b1,P5I);
          a[0]=0;b[0]=0;
          for(i=0;i<nbm/2;i++){
            ip=i*sdva[l-SB];
            if(y[i][0]<0){
              y[i][0]*=(-1);
              dodaj(b,&y[i][0],ip);
            }
            else dodaj(a,&y[i][0],ip);
          }
          ip=(nbm/2)*sdva[l-SB];
          dodaj(a,a1,ip);dodaj(b,b1,ip);
          receiveu(a1,P3I);receiveu(b1,P3I);
          ip=nbm*sdva[l-SB];
          dodaj(a,a1,ip);dodaj(b,b1,ip);
          sendu(a,P0O);sendu(b,P0O);
          n=ChanInInt(P0I); /* ceka pocetak novog kvadriranja... */
        }
        ChanOutInt(P3O,0);ChanOutInt(P5O,0);
      }/* main */



      /* UP2.C :
            Program koji izvrsava procesor P2
      */
      #include <stdio.h>
      #include <conc.h>
      #include "uzag.h"
      #include "zp2.h"
      int sdva[21],t[TKX][TKY];
      main(){
        int n,i,j,l,k,sfi,somg,a0,m,dnl,bm,mb,nbm,ip,it,x0;
        int static a[DUZ_A],b[DUZ_A],a1[DUZ_A1],b1[DUZ_A1];
        int static x[DUZ_UI][DUZ_UJ],y[DUZ_UI][DUZ_UJ];
        inic();
        n=ChanInInt(P0I);
        while(n!=0){ /* kada primi 0 onda se zaustavlja proces kvadriranja */
          /* razmena podataka */
          ChanOutInt(P6O,n);
          l=n/2;k=n-l;
          sfi=sdva[l+1-k];somg=2*sfi;
          dnl=sdva[l-SB];m=l+1;
          bm=sdva[k];mb=bm-1;
          a0=ChanInInt(P0I);
          primi(2,a0,dnl,4,a,P0I);
          ChanOutInt(P6O,a[0]);
          predaj(1,a[0],dnl,2,a,P6O);
          pp(k,2,dnl,a[0],2,sfi,m,a,x);/* pripremna faza tj. FT nad
                                  odgovarajucim podacima   */
          nbm=sdva[k-SBP];
          /* razmena podataka sa procesorom P6 ... u cilju dobijanja
            FT vecih podnizova   */
          for(i=0;i<nbm/8;i++){
            sendu(&x[8*i+4][0],P6O);sendu(&x[8*i+5][0],P6O);
            sendu(&x[8*i+6][0],P6O);sendu(&x[8*i+7][0],P6O);
          }
          for(i=0;i<nbm/8;i++){
            receiveu(&y[4*i][0],P6I);receiveu(&y[4*i+1][0],P6I);
            receiveu(&y[4*i+2][0],P6I);receiveu(&y[4*i+3][0],P6I);
```

```
    }
    for(i=0;i<nbm/8;i++){
      abc2(&x[8*i][0],&y[4*i][0],&y[4*i+nbm/2][0],somg*4*(8*i),m);
      abc2(&x[8*i+1][0],&y[4*i+1][0],&y[4*i+1+nbm/2][0],somg*4*(8*i+1),m);
      abc2(&x[8*i+2][0],&y[4*i+2][0],&y[4*i+2+nbm/2][0],somg*4*(8*i+2),m);
      abc2(&x[8*i+3][0],&y[4*i+3][0],&y[4*i+3+nbm/2][0],somg*4*(8*i+3),m);
    }
    /* razmena podataka sa procesorom P0 ... u cilju dobijanja
       FT vecih podnizova   */
    for(i=0;i<nbm/4;i++){
      receiveu(&x[2*i][0],P0I);receiveu(&x[2*i+1][0],P0I);
    }
    for(i=0;i<nbm/4;i++){
      sendu(&y[4*i][0],P0O);sendu(&y[4*i+1][0],P0O);
    }
    for(i=0;i<nbm/4;i++){
     abc1(&x[2*i][0],&y[4*i+2][0],&x[2*i+nbm/2][0],somg*2*(8*i+2),m);
     abc1(&x[2*i+1][0],&y[4*i+3][0],&x[2*i+1+nbm/2][0],somg*2*(8*i+3),m);
    }
    /* razmena podataka sa procesorom P3 ... u cilju dobijanja
       FT vecih podnizova   */
    for(i=0;i<nbm/2;i++)sendu(&x[2*i+1][0],P3O);
    for(i=0;i<nbm/2;i++)receiveu(&y[i][0],P3I);
    for(i=0;i<nbm/2;i++){
      abc2(&x[2*i][0],&y[i][0],&y[i+nbm/2][0],somg*(8*i+2),m);
    }
    for(i=0;i<nbm;i++){
      it=t[k-SBP][i];x0=y[it][0];
      for(j=0;j<=x0;j++)x[i][j]=y[it][j];
    }
    /* drugi korak ... tj. kvadriranje odgovarajucih komponenti */
    for(i=0;i<nbm;i++){
      kvadriraj(&x[i][0],m);
    }
    /* pocetak inverznih transformacija */
    somg*=(-1);
    ffti(x,k-SBP,m,somg*8);
    for(i=nbm/2;i<nbm;i++)sendu(&x[i][0],P6O);
    for(i=0;i<nbm/2;i++)receiveu(&y[i][0],P6I);
    for(i=0;i<nbm/2;i++)abc1(&x[i][0],&y[i][0],&x[i+nbm/2][0],somg*4*i,m);
    for(i=0;i<nbm/2;i++)receiveu(&y[i][0],P0I);
    for(i=0;i<nbm/2;i++)sendu(&x[i][0],P0O);
    for(i=0;i<nbm/2;i++)
      abc1(&y[i][0],&x[i+nbm/2][0],&y[i+nbm/2][0],somg*2*(nbm+i),m);
    for(i=nbm/2;i<nbm;i++)sendu(&y[i][0],P3O);
    for(i=0;i<nbm/2;i++)receiveu(&x[i][0],P3I);
    for(i=0;i<nbm/2;i++)
      abc2(&y[i][0],&x[i][0],&x[i+nbm/2][0],somg*(i+nbm),m);
    for(i=0;i<nbm/2;i++){
      pomeri(&x[i][0],-(k+sfi*(i+nbm)),m);
      pomeri(&x[i+nbm/2][0],-(k+sfi*(i+nbm+bm/2)),m);
    }
    /* prosledjivanje podataka od procesora PH koji je izvrsio 4. korak */
    for(i=0;i<nbm;i++){
      a1[i]=ChanInInt(P0I);
    }
    for(i=nbm/2;i<nbm;i++){
      ChanOutInt(P6O,a1[i]);
    }
    for(i=nbm;i<2*nbm;i++){
      a1[i]=ChanInInt(P0I);
    }
    for(i=3*nbm/2;i<2*nbm;i++){
```

```
            ChanOutInt(P6O,a1[i]);
          }
          for(i=0;i<nbm/2;i++){
            if(x[i][0]>0)a1[i]=(a1[i]-(x[i][1]&mb)+bm)&mb;
            if(x[i+nbm/2][0]>0){
              a1[i+nbm/2]=(a1[i+nbm]-(x[i+nbm/2][1]&mb)+bm)&mb;
            }
            else a1[i+nbm/2]=a1[i+nbm];
          }
          /* peti korak... */
          for(i=0;i<nbm/2;i++){
            sracunaj(a1[i],i+nbm,k,m,&x[i][0]);
            sracunaj(a1[i+nbm/2],i+nbm+bm/2,k,m,&x[i+nbm/2][0]);
          }
          /* sesti korak... tj. sumiranje */
          a1[0]=0;b1[0]=0;
          for(i=0;i<nbm/2;i++){
            ip=i*sdva[l-SB];
            if(x[nbm/2+i][0]<0){
              x[nbm/2+i][0]*=(-1);
              dodaj(b1,&x[nbm/2+i][0],ip);
            }
            else dodaj(a1,&x[nbm/2+i][0],ip);
          }
          sendu(a1,P6O);sendu(b1,P6O);
          receiveu(a1,P6I);receiveu(b1,P6I);
          a[0]=0;b[0]=0;
          for(i=0;i<nbm/2;i++){
            ip=i*sdva[l-SB];
            if(x[i][0]<0){
              x[i][0]*=(-1);
              dodaj(b,&x[i][0],ip);
            }
            else dodaj(a,&x[i][0],ip);
          }
          ip=(nbm/2)*sdva[l-SB];
          dodaj(a,a1,ip);dodaj(b,b1,ip);
          sendu(a,P0O);sendu(b,P0O);
          n=ChanInInt(P0I); /* ceka pocetak novog kvadriranja... */
        }
        ChanOutInt(P6O,0);
      }/* main */



      /* UP3.C :
           Program koji izvrsava procesor P3
      */
      #include <stdio.h>
      #include <conc.h>
      #include "uzag.h"
      #include "zp3.h"
      int sdva[21],t[TKX][TKY];
      main(){
        int n,i,j,l,k,sfi,somg,a0,m,dnl,bm,mb,nbm,ip,it,x0;
        int static a[DUZ_A],b[DUZ_A],a1[DUZ_A1],b1[DUZ_A1];
        int static x[DUZ_UI][DUZ_UJ],y[DUZ_UI][DUZ_UJ];
        inic();
        n=ChanInInt(P1I);
        while(n!=0){ /* kada primi 0 onda se zaustavlja proces kvadriranja */
          /* razmena podataka */
          ChanOutInt(P7O,n);
          l=n/2;k=n-l;
```

```
    sfi=sdva[l+1-k];somg=2*sfi;
    dnl=sdva[l-SB];m=l+1;
    bm=sdva[k];mb=bm-1;
    a0=ChanInInt(P1I);
    primi(1,a0,dnl,2,a,P1I);
    ChanOutInt(P7O,a[0]);
    predaj(1,a[0],dnl,2,a,P7O);
    pp(k,2,dnl,a[0],3,sfi,m,a,x);/* pripremna faza tj. FT nad
                        odgovarajucim podacima   */
    nbm=sdva[k-SBP];
    /* razmena podataka sa procesorom P7 ... u cilju dobijanja
      FT vecih podnizova   */
    for(i=0;i<nbm/8;i++){
       sendu(&x[8*i+4][0],P7O);sendu(&x[8*i+5][0],P7O);
       sendu(&x[8*i+6][0],P7O);sendu(&x[8*i+7][0],P7O);
    }
    for(i=0;i<nbm/8;i++){
       receiveu(&y[4*i][0],P7I);receiveu(&y[4*i+1][0],P7I);
       receiveu(&y[4*i+2][0],P7I);receiveu(&y[4*i+3][0],P7I);
    }
    for(i=0;i<nbm/8;i++){
       abc2(&x[8*i][0],&y[4*i][0],&y[4*i+nbm/2][0],somg*4*(8*i),m);
       abc2(&x[8*i+1][0],&y[4*i+1][0],&y[4*i+1+nbm/2][0],somg*4*(8*i+1),m);
       abc2(&x[8*i+2][0],&y[4*i+2][0],&y[4*i+2+nbm/2][0],somg*4*(8*i+2),m);
       abc2(&x[8*i+3][0],&y[4*i+3][0],&y[4*i+3+nbm/2][0],somg*4*(8*i+3),m);
    }
    /* razmena podataka sa procesorom P1 ... u cilju dobijanja
      FT vecih podnizova   */
    for(i=0;i<nbm/4;i++){
       receiveu(&x[2*i][0],P1I);receiveu(&x[2*i+1][0],P1I);
    }
    for(i=0;i<nbm/4;i++){
       sendu(&y[4*i][0],P1O);sendu(&y[4*i+1][0],P1O);
    }
    for(i=0;i<nbm/4;i++){
      abc1(&x[2*i][0],&y[4*i+2][0],&x[2*i+nbm/2][0],somg*2*(8*i+2),m);
      abc1(&x[2*i+1][0],&y[4*i+3][0],&x[2*i+1+nbm/2][0],somg*2*(8*i+3),m);
    }
    /* razmena podataka sa procesorom P2 ... u cilju dobijanja
      FT vecih podnizova   */
   for(i=0;i<nbm/2;i++)receiveu(&y[i][0],P2I);
   for(i=0;i<nbm/2;i++)sendu(&x[2*i][0],P2O);
   for(i=0;i<nbm/2;i++){
      abc1(&y[i][0],&x[2*i+1][0],&y[i+nbm/2][0],somg*(8*i+3),m);
   }
   for(i=0;i<nbm;i++){
      it=t[k-SBP][i];x0=y[it][0];
      for(j=0;j<=x0;j++)x[i][j]=y[it][j];
   }
   /* drugi korak ... tj. kvadriranje odgovarajucih komponenti */
   for(i=0;i<nbm;i++){
      kvadriraj(&x[i][0],m);
   }
   /* pocetak inverznih transformacija */
   somg*=(-1);
   ffti(x,k-SBP,m,somg*8);
   for(i=nbm/2;i<nbm;i++)sendu(&x[i][0],P7O);
   for(i=0;i<nbm/2;i++)receiveu(&y[i][0],P7I);
   for(i=0;i<nbm/2;i++)abc1(&x[i][0],&y[i][0],&x[i+nbm/2][0],somg*4*i,m);
   for(i=0;i<nbm/2;i++)receiveu(&y[i][0],P1I);
   for(i=0;i<nbm/2;i++)sendu(&x[i][0],P1O);
   for(i=0;i<nbm/2;i++)
      abc1(&y[i][0],&x[i+nbm/2][0],&y[i+nbm/2][0],somg*2*(nbm+i),m);
```

```
            for(i=0;i<nbm/2;i++)receiveu(&x[i][0],P2I);
            for(i=0;i<nbm/2;i++)sendu(&y[i][0],P2O);
            for(i=0;i<nbm/2;i++)
                abc1(&x[i][0],&y[i+nbm/2][0],&x[i+nbm/2][0],somg*(i+3*nbm),m);
            for(i=0;i<nbm/2;i++){
              pomeri(&x[i][0],-(k+sfi*(i+3*nbm)),m);
              pomeri(&x[i+nbm/2][0],-(k+sfi*(i+3*nbm+bm/2)),m);
            }
            /* prosledjivanje podataka od procesora PH koji je izvrsio 4. korak */
            for(i=0;i<nbm;i++){
              a1[i]=ChanInInt(P1I);
            }
            for(i=nbm/2;i<nbm;i++){
              ChanOutInt(P7O,a1[i]);
            }
            for(i=nbm;i<2*nbm;i++){
              a1[i]=ChanInInt(P1I);
            }
            for(i=3*nbm/2;i<2*nbm;i++){
              ChanOutInt(P7O,a1[i]);
            }
            for(i=0;i<nbm/2;i++){
              if(x[i][0]>0)a1[i]=(a1[i]-(x[i][1]&mb)+bm)&mb;
              if(x[i+nbm/2][0]>0){
                a1[i+nbm/2]=(a1[i+nbm]-(x[i+nbm/2][1]&mb)+bm)&mb;
              }
              else a1[i+nbm/2]=a1[i+nbm];
            }
            /* peti korak... */
            for(i=0;i<nbm/2;i++){
              sracunaj(a1[i],i+3*nbm,k,m,&x[i][0]);
              sracunaj(a1[i+nbm/2],i+3*nbm+bm/2,k,m,&x[i+nbm/2][0]);
            }
            /* sesti korak... tj. sumiranje */
            a1[0]=0;b1[0]=0;
            for(i=0;i<nbm/2;i++){
              ip=i*sdva[l-SB];
              if(x[nbm/2+i][0]<0){
                x[nbm/2+i][0]*=(-1);
                dodaj(b1,&x[nbm/2+i][0],ip);
              }
              else dodaj(a1,&x[nbm/2+i][0],ip);
            }
            sendu(a1,P7O);sendu(b1,P7O);
            receiveu(a1,P7I);receiveu(b1,P7I);
            a[0]=0;b[0]=0;
            for(i=0;i<nbm/2;i++){
              ip=i*sdva[l-SB];
              if(x[i][0]<0){
                x[i][0]*=(-1);
                dodaj(b,&x[i][0],ip);
              }
              else dodaj(a,&x[i][0],ip);
            }
            ip=(nbm/2)*sdva[l-SB];
            dodaj(a,a1,ip);dodaj(b,b1,ip);
            sendu(a,P1O);sendu(b,P1O);
            n=ChanInInt(P1I); /* ceka pocetak novog kvadriranja... */
          }
          ChanOutInt(P7O,0);
        }/* main */
```

```
/* UP4.C :
      Program koji izvrsava procesor P4
*/
#include <stdio.h>
#include <conc.h>
#include "uzag.h"
#include "zp4.h"
int sdva[21],t[TKX][TKY];
main(){
  int n,i,j,l,k,sfi,somg,a0,m,dnl,bm,mb,nbm,ip,it,x0;
  static int a[DUZ_A],b[DUZ_A],a1[DUZ_A1],b1[DUZ_A1];
  static int x[DUZ_UI][DUZ_UJ],y[DUZ_UI][DUZ_UJ];
  inic();
  n=ChanInInt(P0I);
  while(n!=0){ /* kada primi 0 onda se zaustavlja proces kvadriranja */
    /* razmena podataka */
    l=n/2;k=n-l;
    sfi=sdva[l+1-k];somg=2*sfi;
    dnl=sdva[l-SB];m=l+1;
    bm=sdva[k];mb=bm-1;
    a0=ChanInInt(P0I);
    primi(4,a0,dnl,8,a,P0I);
    pp(k,l,dnl,a[0],4,sfi,m,a,x);/* pripremna faza tj. FT nad
                          odgovarajucim podacima   */
    nbm=sdva[k-SBP];
    /* razmena podataka sa procesorom P0 ... u cilju dobijanja
      FT vecih podnizova   */
    for(i=0;i<nbm/8;i++){
      receiveu(&y[4*i][0],P0I);receiveu(&y[4*i+1][0],P0I);
      receiveu(&y[4*i+2][0],P0I);receiveu(&y[4*i+3][0],P0I);
    }
    for(i=0;i<nbm/8;i++){
      sendu(&x[8*i][0],P0O);sendu(&x[8*i+1][0],P0O);
      sendu(&x[8*i+2][0],P0O);sendu(&x[8*i+3][0],P0O);
    }
    for(i=0;i<nbm/8;i++){
     abc1(&y[4*i][0],&x[8*i+4][0],&y[4*i+nbm/2][0],somg*4*(8*i+4),m);
     abc1(&y[4*i+1][0],&x[8*i+5][0],&y[4*i+1+nbm/2][0],somg*4*(8*i+5),m);
     abc1(&y[4*i+2][0],&x[8*i+6][0],&y[4*i+2+nbm/2][0],somg*4*(8*i+6),m);
     abc1(&y[4*i+3][0],&x[8*i+7][0],&y[4*i+3+nbm/2][0],somg*4*(8*i+7),m);
    }
    /* razmena podataka sa procesorom P6 ... u cilju dobijanja
      FT vecih podnizova   */
    for(i=0;i<nbm/4;i++){
      sendu(&y[4*i+2][0],P6O);sendu(&y[4*i+3][0],P6O);
    }
    for(i=0;i<nbm/4;i++){
      receiveu(&x[2*i][0],P6I);receiveu(&x[2*i+1][0],P6I);
    }
    for(i=0;i<nbm/4;i++){
      abc2(&y[4*i][0],&x[2*i][0],&x[2*i+nbm/2][0],somg*2*(8*i+4),m);
      abc2(&y[4*i+1][0],&x[2*i+1][0],&x[2*i+1+nbm/2][0],somg*2*(8*i+5),m);
    }
    /* razmena podataka sa procesorom P5 ... u cilju dobijanja
      FT vecih podnizova   */
    for(i=0;i<nbm/2;i++)sendu(&x[2*i+1][0],P5O);
    for(i=0;i<nbm/2;i++)receiveu(&y[i][0],P5I);
    for(i=0;i<nbm/2;i++){
      abc2(&x[2*i][0],&y[i][0],&y[i+nbm/2][0],somg*(8*i+4),m);
    }
    for(i=0;i<nbm;i++){
```

```
            it=t[k-SBP][i];x0=y[it][0];
            for(j=0;j<=x0;j++)x[i][j]=y[it][j];
          }
          /* drugi korak ... tj. kvadriranje odgovarajucih komponenti */
          for(i=0;i<nbm;i++){
            kvadriraj(&x[i][0],m);
          }
          /* pocetak inverznih transformacija */
          somg*=(-1);
          ffti(x,k-SBP,m,somg*8);
          for(i=0;i<nbm/2;i++)receiveu(&y[i][0],P0I);
          for(i=0;i<nbm/2;i++)sendu(&x[i][0],P0O);
          for(i=0;i<nbm/2;i++)
            abc1(&y[i][0],&x[i+nbm/2][0],&y[i+nbm/2][0],somg*4*(i+nbm/2),m);
          for(i=nbm/2;i<nbm;i++)sendu(&y[i][0],P6O);
          for(i=0;i<nbm/2;i++)receiveu(&x[i][0],P6I);
          for(i=0;i<nbm/2;i++)
            abc2(&y[i][0],&x[i][0],&x[i+nbm/2][0],somg*2*(i+nbm/2),m);
          for(i=nbm/2;i<nbm;i++)sendu(&x[i][0],P5O);
          for(i=0;i<nbm/2;i++)receiveu(&y[i][0],P5I);
          for(i=0;i<nbm/2;i++)
            abc1(&x[i][0],&y[i][0],&x[i+nbm/2][0],somg*(i+nbm/2),m);
          for(i=0;i<nbm/2;i++){
            pomeri(&x[i][0],-(k+sfi*(i+nbm/2)),m);
            pomeri(&x[i+nbm/2][0],-(k+sfi*(i+nbm/2+bm/2)),m);
          }
          /* prosledjivanje podataka od procesora PH koji je izvrsio 4. korak */
          for(i=0;i<nbm;i++){
            a1[i]=ChanInInt(P0I);
          }
          for(i=0;i<nbm;i++){
            if(x[i][0]>0)a1[i]=(a1[i]-(x[i][1]&mb)+bm)&mb;
          }
          /* peti korak... */
          for(i=0;i<nbm/2;i++){
            sracunaj(a1[i],i+nbm/2,k,m,&x[i][0]);
            sracunaj(a1[i+nbm/2],i+nbm/2+bm/2,k,m,&x[i+nbm/2][0]);
          }
          /* sesti korak... tj. sumiranje */
          a1[0]=0;b1[0]=0;
          for(i=0;i<nbm/2;i++){
            ip=i*sdva[l-SB];
            if(x[i][0]<0){
              x[i][0]*=(-1);
              dodaj(b1,&x[i][0],ip);
            }
            else dodaj(a1,&x[i][0],ip);
          }
          receiveu(a,P0I);receiveu(b,P0I);
          sendu(a1,P0O);sendu(b1,P0O);
          a1[0]=0;b1[0]=0;
          for(i=0;i<nbm/2;i++){
            ip=i*sdva[l-SB];
            if(x[i+nbm/2][0]<0){
              x[i+nbm/2][0]*=(-1);
              dodaj(b1,&x[i+nbm/2][0],ip);
            }
            else dodaj(a1,&x[i+nbm/2][0],ip);
          }
          ip=(nbm/2)*sdva[l-SB];
          dodaj(a,a1,ip);dodaj(b,b1,ip);
          receiveu(a1,P6I);receiveu(b1,P6I);
          ip=nbm*sdva[l-SB];
```

```
        dodaj(a,a1,ip);dodaj(b,b1,ip);
        receiveu(a1,P5I);receiveu(b1,P5I);
        ip=2*nbm*sdva[l-SB];
        dodaj(a,a1,ip);dodaj(b,b1,ip);
        sendu(a,P0O);sendu(b,P0O);
        n=ChanInInt(P0I); /* ceka pocetak novog kvadriranja... */
    }
}/* main */



/* UP5.C :
      Program koji izvrsava procesor P5
*/
#include <stdio.h>
#include <conc.h>
#include "uzag.h"
#include "zp5.h"
int sdva[21],t[TKX][TKY];
main(){
  int n,i,j,l,k,sfi,somg,a0,m,dnl,bm,mb,nbm,ip,it,x0;
  static int a[DUZ_A],b[DUZ_A],a1[DUZ_A1],b1[DUZ_A1];
  static int x[DUZ_UI][DUZ_UJ],y[DUZ_UI][DUZ_UJ];
  inic();
  n=ChanInInt(P1I);
  while(n!=0){ /* kada primi 0 onda se zaustavlja proces kvadriranja */
    /* razmena podataka */
    l=n/2;k=n-l;
    sfi=sdva[l+1-k];somg=2*sfi;
    dnl=sdva[l-SB];m=l+1;
    bm=sdva[k];mb=bm-1;
    a0=ChanInInt(P1I);
    primi(2,a0,dnl,4,a,P1I);
    pp(k,1,dnl,a[0],5,sfi,m,a,x);/* pripremna faza tj. FT nad
                          odgovarajucim podacima   */
    nbm=sdva[k-SBP];
    /* razmena podataka sa procesorom P1 ... u cilju dobijanja
      FT vecih podnizova   */
    for(i=0;i<nbm/8;i++){
      receiveu(&y[4*i][0],P1I);receiveu(&y[4*i+1][0],P1I);
      receiveu(&y[4*i+2][0],P1I);receiveu(&y[4*i+3][0],P1I);
    }
    for(i=0;i<nbm/8;i++){
      sendu(&x[8*i][0],P1O);sendu(&x[8*i+1][0],P1O);
      sendu(&x[8*i+2][0],P1O);sendu(&x[8*i+3][0],P1O);
    }
    for(i=0;i<nbm/8;i++){
     abc1(&y[4*i][0],&x[8*i+4][0],&y[4*i+nbm/2][0],somg*4*(8*i+4),m);
     abc1(&y[4*i+1][0],&x[8*i+5][0],&y[4*i+1+nbm/2][0],somg*4*(8*i+5),m);
     abc1(&y[4*i+2][0],&x[8*i+6][0],&y[4*i+2+nbm/2][0],somg*4*(8*i+6),m);
     abc1(&y[4*i+3][0],&x[8*i+7][0],&y[4*i+3+nbm/2][0],somg*4*(8*i+7),m);
    }
    /* razmena podataka sa procesorom P7 ... u cilju dobijanja
      FT vecih podnizova   */
    for(i=0;i<nbm/4;i++){
      sendu(&y[4*i+2][0],P7O);sendu(&y[4*i+3][0],P7O);
    }
    for(i=0;i<nbm/4;i++){
      receiveu(&x[2*i][0],P7I);receiveu(&x[2*i+1][0],P7I);
    }
    for(i=0;i<nbm/4;i++){
     abc2(&y[4*i][0],&x[2*i][0],&x[2*i+nbm/2][0],somg*2*(8*i+4),m);
     abc2(&y[4*i+1][0],&x[2*i+1][0],&x[2*i+1+nbm/2][0],somg*2*(8*i+5),m);
```

```
        }
        /* razmena podataka sa procesorom P4 ... u cilju dobijanja
           FT vecih podnizova   */
        for(i=0;i<nbm/2;i++)receiveu(&y[i][0],P4I);
        for(i=0;i<nbm/2;i++)sendu(&x[2*i][0],P4O);
        for(i=0;i<nbm/2;i++){
           abc1(&y[i][0],&x[2*i+1][0],&y[i+nbm/2][0],somg*(8*i+5),m);
        }
        for(i=0;i<nbm;i++){
           it=t[k-SBP][i];x0=y[it][0];
           for(j=0;j<=x0;j++)x[i][j]=y[it][j];
        }
        /* drugi korak ... tj. kvadriranje odgovarajucih komponenti */
        for(i=0;i<nbm;i++){
           kvadriraj(&x[i][0],m);
        }
        /* pocetak inverznih transformacija */
        somg*=(-1);
        ffti(x,k-SBP,m,somg*8);
        for(i=0;i<nbm/2;i++)receiveu(&y[i][0],P1I);
        for(i=0;i<nbm/2;i++)sendu(&x[i][0],P1O);
        for(i=0;i<nbm/2;i++)
           abc1(&y[i][0],&x[i+nbm/2][0],&y[i+nbm/2][0],somg*4*(i+nbm/2),m);
        for(i=nbm/2;i<nbm;i++)sendu(&y[i][0],P7O);
        for(i=0;i<nbm/2;i++)receiveu(&x[i][0],P7I);
        for(i=0;i<nbm/2;i++)
           abc2(&y[i][0],&x[i][0],&x[i+nbm/2][0],somg*2*(i+nbm/2),m);
        for(i=0;i<nbm/2;i++)receiveu(&y[i][0],P4I);
        for(i=0;i<nbm/2;i++)sendu(&x[i][0],P4O);
        for(i=0;i<nbm/2;i++)
           abc1(&y[i][0],&x[i+nbm/2][0],&y[i+nbm/2][0],somg*(i+5*nbm/2),m);
        for(i=0;i<nbm/2;i++){
           pomeri(&y[i][0],-(k+sfi*(i+5*nbm/2)),m);
           pomeri(&y[i+nbm/2][0],-(k+sfi*(i+5*nbm/2+bm/2)),m);
        }
        /* prosledjivanje podataka od procesora PH koji je izvrsio 4. korak */
        for(i=0;i<nbm;i++){
           a1[i]=ChanInInt(P1I);
        }
        for(i=0;i<nbm;i++){
           if(y[i][0]>0)a1[i]=(a1[i]-(y[i][1]&mb)+bm)&mb;
        }
        /* peti korak... */
        for(i=0;i<nbm/2;i++){
           sracunaj(a1[i],i+5*nbm/2,k,m,&y[i][0]);
           sracunaj(a1[i+nbm/2],i+5*nbm/2+bm/2,k,m,&y[i+nbm/2][0]);
        }
        /* sesti korak... tj. sumiranje */
       a1[0]=0;b1[0]=0;
       for(i=0;i<nbm/2;i++){
          ip=i*sdva[l-SB];
          if(y[i][0]<0){
             y[i][0]*=(-1);
             dodaj(b1,&y[i][0],ip);
          }
          else dodaj(a1,&y[i][0],ip);
       }
       receiveu(a,P1I);receiveu(b,P1I);
       sendu(a1,P1O);sendu(b1,P1O);
       a1[0]=0;b1[0]=0;
       for(i=0;i<nbm/2;i++){
          ip=i*sdva[l-SB];
          if(y[i+nbm/2][0]<0){
```

```
            y[i+nbm/2][0]*=(-1);
            dodaj(b1,&y[i+nbm/2][0],ip);
          }
          else dodaj(a1,&y[i+nbm/2][0],ip);
        }
        ip=(nbm/2)*sdva[l-SB];
        dodaj(a,a1,ip);dodaj(b,b1,ip);
        receiveu(a1,P7I);receiveu(b1,P7I);
        ip=nbm*sdva[l-SB];
        dodaj(a,a1,ip);dodaj(b,b1,ip);
        sendu(a,P4O);sendu(b,P4O);
        n=ChanInInt(P1I); /* ceka pocetak novog kvadriranja... */
    }
}/* main */



/* UP6.C :
      Program koji izvrsava procesor P6
*/
#include <stdio.h>
#include <conc.h>
#include "uzag.h"
#include "zp6.h"
int sdva[21],t[TKX][TKY];
main(){
  int n,i,j,l,k,sfi,somg,a0,m,dnl,bm,mb,nbm,ip,it,x0;
  static int a[DUZ_A],b[DUZ_A],a1[DUZ_A1],b1[DUZ_A1];
  static int x[DUZ_UI][DUZ_UJ],y[DUZ_UI][DUZ_UJ];
  inic();
  n=ChanInInt(P2I);
  while(n!=0){ /* kada primi 0 onda se zaustavlja proces kvadriranja */
    /* razmena podataka */
    l=n/2;k=n-l;
    sfi=sdva[l+1-k];somg=2*sfi;
    dnl=sdva[l-SB];m=l+1;
    bm=sdva[k];mb=bm-1;
    a0=ChanInInt(P2I);
    primi(1,a0,dnl,2,a,P2I);
    pp(k,l,dnl,a[0],6,sfi,m,a,x);/* pripremna faza tj. FT nad
                        odgovarajucim podacima   */
    nbm=sdva[k-SBP];
    /* razmena podataka sa procesorom P2 ... u cilju dobijanja
      FT vecih podnizova   */
    for(i=0;i<nbm/8;i++){
      receiveu(&y[4*i][0],P2I);receiveu(&y[4*i+1][0],P2I);
      receiveu(&y[4*i+2][0],P2I);receiveu(&y[4*i+3][0],P2I);
    }
    for(i=0;i<nbm/8;i++){
      sendu(&x[8*i][0],P2O);sendu(&x[8*i+1][0],P2O);
      sendu(&x[8*i+2][0],P2O);sendu(&x[8*i+3][0],P2O);
    }
    for(i=0;i<nbm/8;i++){
      abc1(&y[4*i][0],&x[8*i+4][0],&y[4*i+nbm/2][0],somg*4*(8*i+4),m);
      abc1(&y[4*i+1][0],&x[8*i+5][0],&y[4*i+1+nbm/2][0],somg*4*(8*i+5),m);
      abc1(&y[4*i+2][0],&x[8*i+6][0],&y[4*i+2+nbm/2][0],somg*4*(8*i+6),m);
      abc1(&y[4*i+3][0],&x[8*i+7][0],&y[4*i+3+nbm/2][0],somg*4*(8*i+7),m);
    }
    /* razmena podataka sa procesorom P4 ... u cilju dobijanja
      FT vecih podnizova   */
    for(i=0;i<nbm/4;i++){
      receiveu(&x[2*i][0],P4I);receiveu(&x[2*i+1][0],P4I);
    }
```

```
    for(i=0;i<nbm/4;i++){
      sendu(&y[4*i][0],P4O);sendu(&y[4*i+1][0],P4O);
    }
    for(i=0;i<nbm/4;i++){
     abc1(&x[2*i][0],&y[4*i+2][0],&x[2*i+nbm/2][0],somg*2*(8*i+6),m);
     abc1(&x[2*i+1][0],&y[4*i+3][0],&x[2*i+1+nbm/2][0],somg*2*(8*i+7),m);
    }
    /* razmena podataka sa procesorom P7 ... u cilju dobijanja
       FT vecih podnizova   */
    for(i=0;i<nbm/2;i++)sendu(&x[2*i+1][0],P7O);
    for(i=0;i<nbm/2;i++)receiveu(&y[i][0],P7I);
    for(i=0;i<nbm/2;i++){
      abc2(&x[2*i][0],&y[i][0],&y[i+nbm/2][0],somg*(8*i+6),m);
    }
    for(i=0;i<nbm;i++){
      it=t[k-SBP][i];x0=y[it][0];
      for(j=0;j<=x0;j++)x[i][j]=y[it][j];
    }
    /* drugi korak ... tj. kvadriranje odgovarajucih komponenti */
    for(i=0;i<nbm;i++){
      kvadriraj(&x[i][0],m);
    }
    /* pocetak inverznih transformacija */
    somg*=(-1);
    ffti(x,k-SBP,m,somg*8);
    for(i=0;i<nbm/2;i++)receiveu(&y[i][0],P2I);
    for(i=0;i<nbm/2;i++)sendu(&x[i][0],P2O);
    for(i=0;i<nbm/2;i++)
      abc1(&y[i][0],&x[i+nbm/2][0],&y[i+nbm/2][0],somg*4*(i+nbm/2),m);
    for(i=0;i<nbm/2;i++)receiveu(&x[i][0],P4I);
    for(i=0;i<nbm/2;i++)sendu(&y[i][0],P4O);
    for(i=0;i<nbm/2;i++)
      abc1(&x[i][0],&y[i+nbm/2][0],&x[i+nbm/2][0],somg*2*(i+3*nbm/2),m);
    for(i=nbm/2;i<nbm;i++)sendu(&x[i][0],P7O);
    for(i=0;i<nbm/2;i++)receiveu(&y[i][0],P7I);
    for(i=0;i<nbm/2;i++)
      abc1(&x[i][0],&y[i][0],&x[i+nbm/2][0],somg*(i+3*nbm/2),m);
    for(i=0;i<nbm/2;i++){
      pomeri(&x[i][0],-(k+sfi*(i+3*nbm/2)),m);
      pomeri(&x[i+nbm/2][0],-(k+sfi*(i+3*nbm/2+bm/2)),m);
    }
    /* prosledjivanje podataka od procesora PH koji je izvrsio 4. korak */
    for(i=0;i<nbm;i++){
      a1[i]=ChanInInt(P2I);
    }
    for(i=0;i<nbm;i++){
      if(x[i][0]>0)a1[i]=(a1[i]-(x[i][1]&mb)+bm)&mb;
    }
    /* peti korak... */
    for(i=0;i<nbm/2;i++){
      sracunaj(a1[i],i+3*nbm/2,k,m,&x[i][0]);
      sracunaj(a1[i+nbm/2],i+3*nbm/2+bm/2,k,m,&x[i+nbm/2][0]);
    }
    /* sesti korak... tj. sumiranje */
    a1[0]=0;b1[0]=0;
    for(i=0;i<nbm/2;i++){
      ip=i*sdva[l-SB];
      if(x[i][0]<0){
        x[i][0]*=(-1);
        dodaj(b1,&x[i][0],ip);
      }
      else dodaj(a1,&x[i][0],ip);
    }
```

```
        receiveu(a,P2I);receiveu(b,P2I);
        sendu(a1,P2O);sendu(b1,P2O);
        a1[0]=0;b1[0]=0;
        for(i=0;i<nbm/2;i++){
          ip=i*sdva[l-SB];
          if(x[i+nbm/2][0]<0){
            x[i+nbm/2][0]*=(-1);
            dodaj(b1,&x[i+nbm/2][0],ip);
          }
          else dodaj(a1,&x[i+nbm/2][0],ip);
        }
        ip=(nbm/2)*sdva[l-SB];
        dodaj(a,a1,ip);dodaj(b,b1,ip);
        sendu(a,P4O);sendu(b,P4O);
        n=ChanInInt(P2I); /* ceka pocetak novog kvadriranja... */
      }
    }/* main */



    /* UP7.C :
          Program koji izvrsava procesor P7
    */
    #include <stdio.h>
    #include <conc.h>
    #include "uzag.h"
    #include "zp7.h"
    int sdva[21],t[TKX][TKY];
    main(){
      int n,i,j,l,k,sfi,somg,a0,m,dnl,bm,mb,nbm,ip,it,x0;
      static int a[DUZ_A],b[DUZ_A],a1[DUZ_A1],b1[DUZ_A1];
      static int x[DUZ_UI][DUZ_UJ],y[DUZ_UI][DUZ_UJ];
      inic();
      n=ChanInInt(P3I);
      while(n!=0){ /* kada primi 0 onda se zaustavlja proces kvadriranja */
        /* razmena podataka */
        l=n/2;k=n-l;
        sfi=sdva[l+1-k];somg=2*sfi;
        dnl=sdva[l-SB];m=l+1;
        bm=sdva[k];mb=bm-1;
        a0=ChanInInt(P3I);
        primi(1,a0,dnl,2,a,P3I);
        pp(k,1,dnl,a[0],7,sfi,m,a,x);/* pripremna faza tj. FT nad
                                odgovarajucim podacima   */
        nbm=sdva[k-SBP];
        /* razmena podataka sa procesorom P3 ... u cilju dobijanja
          FT vecih podnizova   */
        for(i=0;i<nbm/8;i++){
          receiveu(&y[4*i][0],P3I);receiveu(&y[4*i+1][0],P3I);
          receiveu(&y[4*i+2][0],P3I);receiveu(&y[4*i+3][0],P3I);
        }
        for(i=0;i<nbm/8;i++){
          sendu(&x[8*i][0],P3O);sendu(&x[8*i+1][0],P3O);
          sendu(&x[8*i+2][0],P3O);sendu(&x[8*i+3][0],P3O);
        }
        for(i=0;i<nbm/8;i++){
          abc1(&y[4*i][0],&x[8*i+4][0],&y[4*i+nbm/2][0],somg*4*(8*i+4),m);
          abc1(&y[4*i+1][0],&x[8*i+5][0],&y[4*i+1+nbm/2][0],somg*4*(8*i+5),m);
          abc1(&y[4*i+2][0],&x[8*i+6][0],&y[4*i+2+nbm/2][0],somg*4*(8*i+6),m);
          abc1(&y[4*i+3][0],&x[8*i+7][0],&y[4*i+3+nbm/2][0],somg*4*(8*i+7),m);
        }
        /* razmena podataka sa procesorom P5 ... u cilju dobijanja
```

```
    FT vecih podnizova    */
  for(i=0;i<nbm/4;i++){
    receiveu(&x[2*i][0],P5I);receiveu(&x[2*i+1][0],P5I);
  }
  for(i=0;i<nbm/4;i++){
    sendu(&y[4*i][0],P5O);sendu(&y[4*i+1][0],P5O);
  }
  for(i=0;i<nbm/4;i++){
   abc1(&x[2*i][0],&y[4*i+2][0],&x[2*i+nbm/2][0],somg*2*(8*i+6),m);
   abc1(&x[2*i+1][0],&y[4*i+3][0],&x[2*i+1+nbm/2][0],somg*2*(8*i+7),m);
  }
  /* razmena podataka sa procesorom P6 ... u cilju dobijanja
    FT vecih podnizova    */
  for(i=0;i<nbm/2;i++)receiveu(&y[i][0],P6I);
  for(i=0;i<nbm/2;i++)sendu(&x[2*i][0],P6O);
  for(i=0;i<nbm/2;i++){
    abc1(&y[i][0],&x[2*i+1][0],&y[i+nbm/2][0],somg*(8*i+7),m);
  }
  for(i=0;i<nbm;i++){
    it=t[k-SBP][i];x0=y[it][0];
    for(j=0;j<=x0;j++)x[i][j]=y[it][j];
  }
  /* drugi korak ... tj. kvadriranje odgovarajucih komponenti */
  for(i=0;i<nbm;i++){
    kvadriraj(&x[i][0],m);
  }
  /* pocetak inverznih transformacija */
  somg*=(-1);
  ffti(x,k-SBP,m,somg*8);
  for(i=0;i<nbm/2;i++)receiveu(&y[i][0],P3I);
  for(i=0;i<nbm/2;i++)sendu(&x[i][0],P3O);
  for(i=0;i<nbm/2;i++)
    abc1(&y[i][0],&x[i+nbm/2][0],&y[i+nbm/2][0],somg*4*(i+nbm/2),m);
  for(i=0;i<nbm/2;i++)receiveu(&x[i][0],P5I);
  for(i=0;i<nbm/2;i++)sendu(&y[i][0],P5O);
  for(i=0;i<nbm/2;i++)
    abc1(&x[i][0],&y[i+nbm/2][0],&x[i+nbm/2][0],somg*2*(i+3*nbm/2),m);
  for(i=0;i<nbm/2;i++)receiveu(&y[i][0],P6I);
  for(i=0;i<nbm/2;i++)sendu(&x[i][0],P6O);
  for(i=0;i<nbm/2;i++)
    abc1(&y[i][0],&x[i+nbm/2][0],&y[i+nbm/2][0],somg*(i+7*nbm/2),m);
  for(i=0;i<nbm/2;i++){
    pomeri(&y[i][0],-(k+sfi*(i+7*nbm/2)),m);
    pomeri(&y[i+nbm/2][0],-(k+sfi*(i+7*nbm/2+bm/2)),m);
  }
  /* prosledjivanje podataka od procesora PH koji je izvrsio 4. korak */
  for(i=0;i<nbm;i++){
    a1[i]=ChanInInt(P3I);
  }
  for(i=0;i<nbm;i++){
    if(y[i][0]>0)a1[i]=(a1[i]-(y[i][1]&mb)+bm)&mb;
  }
  /* peti korak... */
  for(i=0;i<nbm/2;i++){
    sracunaj(a1[i],i+7*nbm/2,k,m,&y[i][0]);
    sracunaj(a1[i+nbm/2],i+7*nbm/2+bm/2,k,m,&y[i+nbm/2][0]);
  }
  /* sesti korak... tj. sumiranje */
  a1[0]=0;b1[0]=0;
  for(i=0;i<nbm/2;i++){
    ip=i*sdva[l-SB];
    if(y[i][0]<0){
      y[i][0]*=(-1);
```

```
                dodaj(b1,&y[i][0],ip);
              }
              else dodaj(a1,&y[i][0],ip);
            }
            receiveu(a,P3I);receiveu(b,P3I);
            sendu(a1,P3O);sendu(b1,P3O);
            a1[0]=0;b1[0]=0;
            for(i=0;i<nbm/2;i++){
              ip=i*sdva[I-SB];
              if(y[i+nbm/2][0]<0){
                y[i+nbm/2][0]*=(-1);
                dodaj(b1,&y[i+nbm/2][0],ip);
              }
              else dodaj(a1,&y[i+nbm/2][0],ip);
            }
            ip=(nbm/2)*sdva[I-SB];
            dodaj(a,a1,ip);dodaj(b,b1,ip);
            sendu(a,P5O);sendu(b,P5O);
            n=ChanInInt(P3I); /* ceka pocetak novog kvadriranja... */
          }
        }/* main */


        /* Za pretvaranje sesnestobitnih u osmobitne. */
        #include <stdio.h>
        #define DUZ_A 4097
        #define MASK8 0xff

        void uzmi(int *a,FILE *fp,int *dd){
          int i,j,t;
          i=0;
          fscanf(fp," %d %d\n",&a[0],&t);
          *dd=t;
          while(i<a[0]){
            j=0;
            while((j<12) && (i<a[0])){
              j++;i++;
              fscanf(fp," %x",&a[i]);
            }
            fscanf(fp,"\n");
          }
        }/* uzmi */

        void smesti(int *a,FILE *fp,int dd){
          int i,j;
          i=0;
          fprintf(fp," %d %d\n",a[0],dd);
          while(i<a[0]){
            j=0;
            while((j<12) && (i<a[0])){
              j++;i++;
              fprintf(fp," %x",a[i]);
            }
            fprintf(fp,"\n");
          }
        } /* smesti */


        main(){
          static int a[2*DUZ_A],b[4*DUZ_A];
          int i,n,dd;
          FILE *fp,*fp1;
```

```
        char *imed;
        imed="fc14r.dat";
        fp1=fopen(imed,"r+");
        uzmi(a,fp1,&dd);
        fclose(fp1);
        b[0]=2*a[0];
        for(i=1;i<=a[0];i++){
                b[2*i-1]=a[i] & MASK8;
                b[2*i]=(a[i] >> 8) & MASK8;
        }
        if(b[b[0]]==0)b[0]--;
        fp=fopen("ff15.dat","w+");
        smesti(b,fp1,dd);
        fclose(fp);
        printf("\n KRAJ \n");getchar();
      }/* main */



    /*
      MODF.C  program koji uzima veliki broj R_n iz odgovarajuce datoteke
              i nalazi  R_n po modulu 2^35 - 1 ,  2^36  i  2^36 - 1 .
              Napomenimo da je R_n =3^{2^{2^n -1}} (mod F_n), gdje je F_n
              n-ti Fermaov broj ( tj. F_n = 2^{2^n} +1 ).
              U zavisnosti od definicije vrednosti POJEDIN pruza se mogucnost
              izbora odgovarajuce vrednosti za n, ili izbor odgovarajuceg
              intervala za n.
    */
    #include <stdio.h>
    #define POJEDIN 0   /* ako je 0 onda se n bira iz intervala ... */
    #define DUZ_A 4097 /* broj reci za pamcenje velikog broja tj. R_n */
    #define M1 0x1   /* maske */
    #define M2 0x3
    #define M3 0x7
    #define M4 0xf
    #define M5 0x1f
    #define M8 0xff
    #define DEB 100  /* dekadna baza */

    /*
     UZMI: Ucitava veliki broj iz datoteke i smesta u niz.
           Pretpostavka je da se u jednoj reci pamti osmobitni broj.
    */
    void uzmi(int *a,FILE *fp,int *dd){
      int i,j,t;
      i=0;
      fscanf(fp," %d %d\n",&a[0],&t);
      *dd=t;
      while(i<a[0]){
        j=0;
        while((j<12) && (i<a[0])){
          j++;i++;
          fscanf(fp," %x",&a[i]);
        }
        fscanf(fp,"\n");
      }
    }/* uzmi */
    /*
    void smesti(int *a,FILE *fp,int dd){
      int i,j;
      i=0;
      fprintf(fp," %d %d\n",a[0],dd);
      while(i<a[0]){
```

```
        j=0;
        while((j<12) && (i<a[0])){
          j++;i++;
          fprintf(fp," %x",a[i]);
        }
        fprintf(fp,"\n");
      }
    } smesti
    */

    /*
      DEK: Pretvara broj iz heksadecimalnog zapisa u dekadni
           i prikazuje rezultat
    */
    void dek(int *b,FILE *fp1){
      int i,j,c[12],c0;
      c[0]=1;c[1]=b[9];
      for(j=8;j>0;j--){
          c0=c[0];
          for(i=1;i<=c0;i++)c[i]*=16;
          c[c0+1]=0;
          for(i=1;i<=c0;i++){
            if(c[i]>=DEB){
                c[i+1]+=(c[i]/DEB);
                c[i]=(c[i]%DEB);
            }
          }
          if(c[c0+1]!=0)c[0]++;
          c0=c[0];c[1]+=b[j];c[c0+1]=0;
          for(i=1;i<=c0;i++){
            if(c[i]>=DEB){
                c[i+1]+=(c[i]/DEB);
                c[i]=(c[i]%DEB);
            }
          }
          if(c[c0+1]!=0)c[0]++;
      }
      printf("%d",c[c[0]]);
      for(i=c[0]-1;i>0;i--){
          if(c[i]<10)printf("0%d",c[i]);
          else printf("%d",c[i]);
      }
      printf("   ");
      fprintf(fp1,"%d",c[c[0]]);
      for(i=c[0]-1;i>0;i--){
          if(c[i]<10)fprintf(fp1,"0%d",c[i]);
          else fprintf(fp1,"%d",c[i]);
      }
      fprintf(fp1,"   ");
    }/* dek */

    /* MODF1: Nalazi vrednost ucitanog broja po modulu 2^35-1   */
    void modf1(int *a,FILE *fp1){
      int i,j,a0,r,q,carry,nc,b[80],c[80],c0;
      a0=a[0];b[0]=0;c[0]=0;b[36]=0;
      r=a0%35;q=a0/35;
      for(i=1;i<=r;i++)b[i]=a[35*q+i];
      for(i=r+1;i<=35;i++)b[i]=0;
      for(i=0;i<q;i++){
          for(j=1;j<=35;j++){
            b[j]+=a[35*i+j];
          }
      }
```

```
    carry=0;
    for(i=1;i<=3;i++){
        for(j=1;j<=35;j++){
            nc=(b[j] >> 8) & M8;
            b[j]=(b[j] & M8)+carry;
            carry=nc;
        }
    }
    for(i=0;i<7;i++){
        c[8*i+1]=(b[5*i+1] & M5);
        c[8*i+2]=(((b[5*i+2] << 3) & M5) + ((b[5*i+1] >> 5) & M3));
        c[8*i+3]=((b[5*i+2] >> 2) & M5);
        c[8*i+4]=(((b[5*i+3] << 1) & M5) + ((b[5*i+2] >> 7) & M1));
        c[8*i+5]=(((b[5*i+4] << 4) & M5) + ((b[5*i+3] >> 4) & M4));
        c[8*i+6]=((b[5*i+4] >> 1) & M5);
        c[8*i+7]=(((b[5*i+5] << 2) & M5) + ((b[5*i+4] >> 6) & M2));
        c[8*i+8]=((b[5*i+5] >> 3) & M5);
    }
    for(j=1;j<=7;j++)b[j]=0;
    for(i=0;i<8;i++){
        for(j=1;j<=7;j++){
            b[j]+=c[7*i+j];
        }
    }
    carry=0;
    for(i=1;i<=3;i++){
        for(j=1;j<=7;j++){
            nc=(b[j] >> 5) & M5;
            b[j]=(b[j] & M5)+carry;
            carry=nc;
        }
    }
    carry=0;b[8]=0;
    for(j=1;j<9;j++){
        b[j]=(b[j] << (j-1))+carry;
        carry=(b[j] >> 4);
        b[j]=(b[j] & M4);
    }
    b[9]=carry;
    dek(b,fp1);
}/* modf1 */

/* MODF2: Nalazi vrednost ucitanog broja po modulu 2^36   */
void modf2(int *a,FILE *fp1){
    int i,j,b[12];
    for(i=1;i<=5;i++){
        b[2*i-1]=a[i] & M4;
        b[2*i]=(a[i] >> 4) & M4;
    }
    dek(b,fp1);
}/* modf2 */

/* MODF3: Nalazi vrednost ucitanog broja po modulu 2^36-1   */
void modf3(int *a,FILE *fp1){
    int i,j,a0,r,q,carry,nc,b[80],c[80],c0;
    a0=a[0];b[0]=0;c[0]=0;b[37]=0;
    r=a0%36;q=a0/36;
    for(i=1;i<=r;i++)b[i]=a[36*q+i];
    for(i=r+1;i<=36;i++)b[i]=0;
    for(i=0;i<q;i++){
        for(j=1;j<=36;j++){
            b[j]+=a[36*i+j];
```

```
            }
        }
        carry=0;
        for(i=1;i<=3;i++){
                for(j=1;j<=36;j++){
                    nc=(b[j] >> 8) & M8;
                    b[j]=(b[j] & M8)+carry;
                    carry=nc;
                }
        }
        for(i=1;i<=36;i++){
                c[2*i-1]=b[i] & M4;
                c[2*i]=(b[i] >> 4) & M4;
        }
        c[73]=c[74]=0;
        for(j=1;j<=9;j++)b[j]=0;
        for(i=0;i<8;i++){
                for(j=1;j<=9;j++){
                    b[j]+=c[9*i+j];
                }
        }
        carry=0;
        for(i=1;i<=3;i++){
                for(j=1;j<=9;j++){
                    nc=(b[j] >> 4) & M4;
                    b[j]=(b[j] & M4)+carry;
                    carry=nc;
                }
        }
        dek(b,fp1);
}/* modf3 */

/*
 MAIN: Iz svake datoteke ucita odgovarajuci broj i prikaze
        odgovarajuca  tri modula
*/
main(){
 static int a[2*DUZ_A];
 int n,dd,kraj,poc;
 FILE *fp,*fp1;
 char *imed;
 imed="tr00d.dat";
 fp1=fopen("rmod.dat","w+");
 #if POJEDIN
  printf("\n Unesi n= ");
  scanf("%d",&n);getchar();
  poc=n;kraj=n;
 #else
  printf("\n Unesi gornju granicu: n= ");scanf("%d",&kraj);getchar();
  printf("\n Radim za n =5,6,..., %d \n",kraj);
  poc=5;
 #endif
 for(n=poc;n<=kraj;n++){
  if(n<10){
        imed[3]=48+n;
  }
  else{
        imed[2]=49;imed[3]=38+n;
  }
  fp=fopen(imed,"r+");
  uzmi(a,fp,&dd);
  fclose(fp);
  printf("\n  n=%d   ",n);
```

```
    fprintf(fp1,"\n  n=%d   ",n);
    modf1(a,fp1);modf2(a,fp1);modf3(a,fp1);
  }
  fclose(fp1);
  printf("\n KRAJ \n");getchar();
}/* main */
```

Dobijeni Selfridge-Hurwitzovi ostaci

n=5	10324303	10324303	10324303
n=6	9190530327	8845352501	9017941414
n=7	5799525263	3909272836	44591026080
n=8	30627284506	46310188723	35403253324
n=9	28173182079	19661770102	54966870189
n=10	28022031617	36399120536	54182679152
n=11	3934743084	66666487080	44928212591
n=12	5300454051	64546579219	3387502849
n=13	3434508623	52529728350	52864871946
n=14	15173315214	54038984522	1986493987
n=15	14110954287	7124011679	42435904961

```
/*
    Pepinov test za Fermaove brojeve...
        U zavisnosti od definicije vrednosti POJEDIN pruza se
    mogucnost izbora vrednosti broja za koji primenjujemo test
    ili se test vrsi za sve Fermaove brojeve F_n, 5<= n <=MAX_N.
    Sve dimenzije zavise od stepena baze ( ST ) i maksimalne
    vrednosti za n ( MAX_N ).
*/
#include <stdio.h>
#include <time.h>
#define POJEDIN 1   /* ako je 0 onda se racuna za sve ... */
#define MAX_N 14 /* Maksimalna vrednost... za koju program moze da radi */
#define SB 3 /* Stepen baze, zavisi or velicine procesorske reci */
#define DZB (1 << (SB)) /* Duzina (tj. broj bita) baze */
#define BAZA (1 << (DZB)) /* Velicina baze... */
/* Dimenzije odgovarajucih nizova */
#define DUZN (1+(4*(1 << (((MAX_N) >> 1)-(SB)+1))))
#define DUZ_A (3*(1 << ((MAX_N) - (SB) -1)))
#define DUZ_UI (1+(1 << ((MAX_N) - (((MAX_N) >> 1)))))
#define DUZ_UJ (DUZN)
#define KZAT (1+((MAX_N) - ((MAX_N) >> 1)))
#define DUZ_OD_T (1 << (KZAT))
int sdva[MAX_N+1],t[DUZ_OD_T];

/* INIC: Vrsi inicijalizaciju nizova  sdva[] i t[]  */
void inic(){
   int i,p;
   p=1;
   for(i=0;i<=MAX_N;i++){
      sdva[i]=p;
      p*=2;
   }
   t[0]=0;
   for(i=0;i<KZAT;i++){
      for(p=0;p<sdva[i];p++){
            t[p]=2*t[p];
            t[p+sdva[i]]=t[p]+1;
      }
   }
}

/* ADD: Sabira date brojeve */
void add(int *a,int *b){
  int a0,min,max,i;
  a0=a[0];min=b[0];
  if(a0<min){
     for(i=a0+1;i<=min;i++)a[i]=b[i];
     max=min;min=a0;
  }
  else max=a0;
  a[max+1]=0;
  for(i=1;i<=min;i++){
     a[i]+=b[i];
     if(a[i]>=BAZA){
        a[i]-=BAZA;
        a[i+1]+=1;
     }
  }
  i=min+1;
  while(a[i]>=BAZA){
     a[i]-=BAZA;i++;
     a[i]+=1;
  }
```

```
        if(a[max+1])
          a[0]=max+1;
        else
          a[0]=max;
        } /* add */

      /* SUB: Nalazi razliku brojeva (a-b), pretpostavlja a>b  */
      void sub(int *a,int *b){
        int a0,b0,i;
        b0=b[0];a0=a[0];
        for(i=1;i<=b0;i++){
          a[i]-=b[i];
          if(a[i]<0){
            a[i]+=BAZA;
            a[i+1]-=1;
          }
        }
        i=b0+1;
        while(a[i]<0){
          a[i]+=BAZA;i++;
          a[i]-=1;
        }
        while((a0>0)&&(a[a0]==0))a0--;
        a[0]=a0;
      } /* sub */

      /* MODUO: Nalazi (a mod F_m)  */
      void moduo(int *a,int m){
        int nv,a0,i;
        nv=sdva[m-SB];
        if(a[0]>nv){
          a0=a[0]-nv;a[1]-=a[nv+1];a[nv+1]=0;a[0]=nv;
          if(a[1]<0){
            a[1]+=BAZA;a[2]-=1;
          }
          for(i=2;i<=a0;i++){
            a[i]-=a[nv+i];
            if(a[i]<0){
              a[i]+=BAZA;
              a[i+1]-=1;
            }
          }
          i=a0+1;
          while((i<=nv)&&(a[i]<0)){
            a[i]+=BAZA;i++;
            a[i]-=1;
          }
          if(a[nv+1]<0){
            a[1]+=1;
            i=1;
            while(a[i]==BAZA){
              a[i]=0;i++;
              a[i]+=1;
            }
            if(a[nv+1]==0){
              a[0]=(-1);a[1]=1;
            }
          }
          a0=a[0];
          while((a0>0)&&(a[a0]==0))a0--;
          a[0]=a0;
        }
      }/* moduo */
```

```
/* SQUARE: Kvadrira dati broj po algoritmu Karacuba & Hofman */
void square(int *p){
  int ap,tp,k,i,u[DUZN],v[DUZN],z[DUZN],u0,z0,v0,p0;
  ap=p[0];
  if(ap<0)ap=(-ap);
  if(ap<5){
    switch(p[0]){
      case 0:
      case 1: p[0]=0;
          if(ap==1){
            tp=p[1]*p[1];
            p[1]=tp & (BAZA-1);
                    p[2]=(tp>>DZB) & (BAZA-1);
            if(p[2])p[0]=2;
            else p[0]=1;
          }
          break;
      case 2: u[0]=p[1];u[1]=p[2];u[2]=u[0]-u[1];
          for(i=0;i<=2;i++){
            tp=u[i]*u[i];
                    v[2*i]=tp&(BAZA-1);v[2*i+1]=(tp>>DZB) & (BAZA-1);
          }
          p[1]=v[0];p[2]=v[0]+v[1]+v[2]-v[4];
          p[3]=v[1]+v[2]+v[3]-v[5];p[4]=v[3];p[5]=0;
          for(i=1;i<=4;i++){
            while(p[i]<0){
              p[i]+=BAZA;p[i+1]--;
            }
            while(p[i]>=BAZA){
              p[i]-=BAZA;p[i+1]++;
            }
          }
          p0=5;
          while((p0>0)&&(p[p0]==0))p0--;
          p[0]=p0;
          break;
      case 3: p[4]=0;
      case 4: u[0]=p[1];u[1]=p[2];u[2]=u[0]-u[1];
          u[3]=p[3];u[4]=p[4];u[5]=u[3]-u[4];
          u[7]=p[2]-p[4];
          if(u[7]==0){
            u[8]=u[6]=p[1]-p[3];
          }
          else{
            if(u[7]<0){
              u[6]=p[3]-p[1];u[7]=-u[7];
            }
            else u[6]=p[1]-p[3];
            if(u[6]<0){
              u[6]+=BAZA;u[7]--;
            }
            u[8]=u[6]-u[7];
          }
          for(i=0;i<=8;i++){
            tp=u[i]*u[i];
                    v[2*i]=tp&(BAZA-1);v[2*i+1]=(tp>>DZB) & (BAZA-1);
          }
          z[1]=v[0]+v[2]-v[4];z[3]=v[0]+v[6]-v[12];
          z[5]=v[2]+v[8]-v[14];z[7]=v[6]+v[8]-v[10];
          z[4]=z[1]+z[7]-(v[12]+v[14]-v[16]);
          u[1]=v[1]+v[3]-v[5];u[3]=v[1]+v[7]-v[13];
          u[5]=v[3]+v[9]-v[15];u[7]=v[7]+v[9]-v[11];
```

```
                u[4]=u[1]+u[7]-(v[13]+v[15]-v[17]);
                p[1]=v[0];p[2]=v[1]+z[1];p[3]=u[1]+v[2]+z[3];
                p[4]=v[3]+u[3]+z[4];p[5]=u[4]+z[5]+v[6];
                p[6]=u[5]+v[7]+z[7];p[7]=u[7]+v[8];p[8]=v[9];
                p[9]=0;
                for(i=1;i<=8;i++){
                  while(p[i]<0){
                    p[i]+=BAZA;p[i+1]--;
                  }
                  while(p[i]>=BAZA){
                    p[i]-=BAZA;p[i+1]++;
                  }
                }
                p0=9;
                while((p0>0)&&(p[p0]==0))p0--;
                p[0]=p0;
                break;
            }
          }
          else{
            if(ap%2){
              ap++;p[ap]=0;
            }
            k=ap/2;
            for(i=1;i<=k;i++){
              u[i]=p[i];v[i]=p[k+i];z[i]=u[i]-v[i];
            }
            u0=k;v0=k;
            while((u0>0) && (u[u0]==0))u0--;
            while((v0>0) && (v[v0]==0))v0--;
            u[0]=u0;v[0]=v0;z0=k;
            while((z0>0) && (z[z0]==0))z0--;
            if(z0){
              if(z[z0]<0)
                for(i=1;i<=z0;i++) z[i]= -z[i];
              for(i=1;i<=z0;i++){
                if(z[i]<0){
                  z[i]+=BAZA;
                  z[i+1]-=1;
                }
              }
              while((z0>0)&&(z[z0]==0))z0--;z[0]=z0;
              square(z);
            }
            else
              z[0]=0;
            square(u);square(v);
            u0=u[0];v0=v[0];
            for(i=1;i<=u0;i++)p[i]=u[i];
            for(i=1;i<=v0;i++)p[i+2*k]=v[i];
            for(i=u0+1;i<=2*k;i++)p[i]=0;
            p0=p[0]=2*k+v0;
            add(u,v);
            sub(u,z);
            u0=u[0];p[p0+1]=0;
            for(i=1;i<=u0;i++){
              p[i+k]+=u[i];
              if(p[i+k]>=BAZA){
                p[i+k]-=BAZA;
                p[i+k+1]+=1;
              }
            }
          }
          i=u0+k+1;
```

```
            while(p[i]>=BAZA){
              p[i]-=BAZA;i++;
              p[i]+=1;
            }
            if(p[p0+1])p[0]+=1;
          }
        } /*square */

        /*
          MULT: Izvrsava cetvrti korak Schonhage Strassenovog algoritma.
                Tj. nalazi konvoluciju niza y[] sa samim sobom po datom
                modulu.
        */
        void mult(int *y,int k,int bmas){
          int s,i,dvas,tris,a[DUZ_UI],b[DUZ_UI],c[DUZ_UI];
          if(k>0){
            s=sdva[k-1];
              for(i=0;i<s;i++){
                a[i]=(y[i] & bmas);b[i]=(y[i+s] & bmas);
                c[i]=((bmas+1+a[i]-b[i]) & bmas);
            }
              mult(a,k-1,bmas);mult(b,k-1,bmas);mult(c,k-1,bmas);
            dvas=2*s;tris=3*s;
            for(i=0;i<s;i++){
                y[i]=(a[i] & bmas);y[i+tris]=(b[i+s] & bmas);
                y[i+s]=((bmas+1+a[i+s]+a[i]+b[i]-c[i]) & bmas);
                y[i+dvas]=((bmas+1+b[i]+a[i+s]+b[i+s]-c[i+s]) & bmas);
            }
          }
          else{
              y[0]=((y[0]*y[0]) & bmas);
            y[1]=0;
          }
        } /* mult */

        /*
          PRETVORI: Datom negativnom broju dodaje F_m, kako bi dobio
                najmanju pozetivnu vrednost po modulu F_m.
        */
        void pretvori(int *b,int m){
          int nv,i,b0,bz1,j;
          if((b[0]==-1)||((b[0]==1)&&(b[1]==1)))b[0]*=(-1);
          else{
            if(b[0]!=0){
              nv=sdva[m-SB];b[1]=1-b[1];
              i=2;b0=b[0];bz1=BAZA-1;
              if(b[1]>=0){
                while(b[i]==0)i++;
                b[i]=BAZA-b[i];i++;
              }
              else{
                b[1]+=BAZA;
              }
              for(j=i;j<=b0;j++)b[j]=bz1-b[j];
              for(j=b0+1;j<=nv;j++)b[j]=bz1;
              b0=nv;
              while((b0>0)&&(b[b0]==0))b0--;b[0]=b0;
            }
          }
        }/* pretvori */

        /* POMERI: Vrsi pomeranje datog broja ( b ) za pom mesta ulevo.
                Tj. mnozenje sa 2^pom . Sve to po modulu F_m.
```

```
*/
void pomeri(int *b,int pom,int m){
   int nv,sq,pom1,b0,mask,i,bz1;
   if((b[0]!=0) && (pom!=0)){
      nv=sdva[m];sq=1;
      while(pom<0)pom+=(2*nv);
      while(pom>=(2*nv))pom-=(2*nv);
      if(pom>=nv){
         pom-=nv;sq=-1;
      }
      pom1=pom%DZB;
      pom/=DZB;b0=b[0];
      if(b0==-1){
         if(sq<0){
            for(i=1;i<=pom;i++)b[i]=0;
            b[pom+1]=sdva[pom1];b[0]=pom+1;
         }
         else{
            if((DZB*pom+pom1)!=0){
               b[1]=0;nv=nv/DZB;
               for(i=2;i<=pom;i++)b[i]=0;
               b[pom+1]=BAZA-sdva[pom1];
               bz1=BAZA-1;
               for(i=pom+2;i<=nv;i++)b[i]=bz1;
               b[0]=nv;b[1]+=1;
            }
         }
      }
      else{
         if(pom1>0){
            b0++;b[b0]=0;mask=sdva[DZB-pom1]-1;
            for(i=b0;i>1;i--){
               b[i]=((b[i]&mask)<<pom1)+(b[i-1]>>(DZB-pom1));
            }
            b[1]=(b[1]&mask)<<pom1;
            if(b[b0]==0)b0--;
         }
         if(pom!=0){
            for(i=b0;i>0;i--)b[i+pom]=b[i];
            for(i=1;i<=pom;i++)b[i]=0;
         }
         b[0]=b0+pom;
         moduo(b,m);
         if(sq<0){
            pretvori(b,m);
         }
      }
   }
}/* pomeri */

/* MODSUB: Nalazi razliku datih brojeva po modulu F_m. */
void modsub(int *a,int *b,int *c,int m){
   int nv,i,j,a0,min,max,c0,bz1,b0;
   nv=sdva[m-SB];
   if((a[0]==-1)||(b[0]==-1)){
      if(a[0]==-1){
         if(b[0]==-1){
            c[0]=0;c[1]=0;
         }
         else{
            if(b[0]==0){
               c[0]=-1;c[1]=1;
            }
```

```
                else{
                  i=1;b0=b[0];
                  while((i<b0)&&(b[i]==0)){
                    c[i]=0;i++;
                  }
                  c[i]=BAZA-b[i];bz1=BAZA-1;
                  for(j=i+1;j<=b0;j++)
                    c[j]=bz1-b[j];
                  for(j=b0+1;j<=nv;j++)
                    c[j]=bz1;
                  c0=nv;
                  while((c0>0)&&(c[c0]==0))c0--;c[0]=c0;
                }
              }
            }
            else{
              if(a[0]==0){
                c[0]=c[1]=1;
              }
              else{
                    a0=a[0];c[1]=a[1]+1;i=1;c[nv+1]=c[a0+1]=a[a0+1]=0;
                while(c[i]>=BAZA){
                  c[i]-=BAZA;i++;
                  c[i]=a[i]+1;
                }
                for(j=i+1;j<=a0+1;j++)c[j]=a[j];
                if(c[nv+1]>0){
                  c[0]=-1;c[1]=1;
                }
                else{
                  if(c[a0+1]>0){
                    c[0]=a0+1;
                  }
                  else c[0]=a0;
                }
              }
            }
          }
        }
        else{
          a0=a[0];min=b[0];
          if(a0<min){
            max=min;min=a0;
          }
          else max=a0;
          for(i=1;i<=min;i++)c[i]=a[i]-b[i];
          if(a0==max){
            for(i=min+1;i<=max;i++)c[i]=a[i];
          }
          else{
            for(i=min+1;i<=max;i++)c[i]=-b[i];
          }
          c0=max;c[c0+1]=0;
          while((c0>0)&&(c[c0]==0))c0--;
          if(c0==0)c[0]=0;
          if(c[c0]<0){
            if((c0==1)&&(c[1]==-1)){
              c[0]=-1;c[1]=1;
            }
            else{
              c[1]+=1;i=1;
              while(c[i]==BAZA){
                c[i]=0;i++;
                c[i]+=1;
```

```
            }
            for(j=i;j<c0;j++){
              if(c[j]<0){
                c[j]+=BAZA;c[j+1]-=1;
              }
            }
            c[c0]+=BAZA;bz1=BAZA-1;
            for(i=c0+1;i<=nv;i++)c[i]=bz1;
            c0=nv;
            while((c0>0)&&(c[c0]==0))c0--;
            c[0]=c0;
          }
        }
        else{
          for(i=1;i<=c0;i++){
            if(c[i]<0){
              c[i]+=BAZA;c[i+1]-=1;
            }
          }
          while((c0>0)&&(c[c0]==0))c0--;
          c[0]=c0;
        }
      }
    } /* modsub */

    /* FFT: Nalazi FT datog niza.  */
    void fft(int x[][DUZ_UJ],int k,int m,int somg){
      int j,sdvkj1,sdvj,sdvj1,s,szdj1,i,p1,p2,q1,xp0,q,tx[2*DUZ_UJ];
      for(j=0;j<k;j++){
        sdvkj1=sdva[k-j-1];sdvj=sdva[j];sdvj1=2*sdvj;
        for(s=0;s<sdvkj1;s++){
          szdj1=s*sdvj1;
            for(i=0;i<sdvj;i++){
                p1=(t[szdj1+i] >> (KZAT-k));
                p2=(t[szdj1+i+sdvj] >> (KZAT-k));
            xp0=x[p2][0];
            for(q=0;q<=xp0;q++)tx[q]=x[p2][q];
            pomeri(tx,somg*i*sdvkj1,m);
            modsub(&x[p1][0],tx,&x[p2][0],m);
            if((x[p1][0]==-1)||(tx[0]==-1)){
              if(tx[0]==-1){
                tx[0]=1;
                modsub(&x[p1][0],tx,&x[p1][0],m);
              }
              else{
                if(tx[0]>0){
                  xp0=tx[0];
                  x[p1][1]=tx[1]-1;q=1;
                  while(x[p1][q]<0){
                    x[p1][q]+=BAZA;q++;
                    x[p1][q]=tx[q]-1;
                  }
                  for(q1=q+1;q1<=xp0;q1++)
                    x[p1][q1]=tx[q1];
                  while((xp0>0)&&(x[p1][xp0]==0))xp0--;
                  x[p1][0]=xp0;
                }
              }
            }
            else{
              add(&x[p1][0],tx);
              modulo(&x[p1][0],m);
            }
```

```
            }
          }
        }
      }/* fft */

      /* FFTI: Nalazi inverzne FT od datog niza. */
      void ffti(int x[][DUZ_UJ],int k,int m,int somg){
        int j,sdvkj1,sdvj,sdvj1,s,szdj1,i,p1,p2,xp0,q,q1,tx[2*DUZ_UJ];
        for(j=0;j<k;j++){
          sdvkj1=sdva[k-j-1];sdvj=sdva[j];sdvj1=2*sdvj;
          for(s=0;s<sdvkj1;s++){
            szdj1=s*sdvj1;
            for(i=0;i<sdvj;i++){
              p1=szdj1+i;
              p2=p1+sdvj;
              xp0=x[p2][0];
              for(q=0;q<=xp0;q++)tx[q]=x[p2][q];
              pomeri(tx,somg*i*sdvkj1,m);
              modsub(&x[p1][0],tx,&x[p2][0],m);
              if((x[p1][0]==-1)||(tx[0]==-1)){
                if(tx[0]==-1){
                  tx[0]=1;
                  modsub(&x[p1][0],tx,&x[p1][0],m);
                }
                else{
                  if(tx[0]>0){
                    xp0=tx[0];
                    x[p1][1]=tx[1]-1;q=1;
                    while(x[p1][q]<0){
                      x[p1][q]+=BAZA;q++;
                      x[p1][q]=tx[q]-1;
                    }
                    for(q1=q+1;q1<=xp0;q1++)
                      x[p1][q1]=tx[q1];
                    while((xp0>0)&&(x[p1][xp0]==0))xp0--;
                    x[p1][0]=xp0;
                  }
                }
              }
              else{
                add(&x[p1][0],tx);
                moduo(&x[p1][0],m);
              }
            }
          }
        }
      }/* ffti */

      /* DODAJ: Broju a dodaje broj y*(BAZA)^(ip)   */
      void dodaj(int *a,int *y,int ip){
        int a0,y0,i,tmp;
        a0=a[0];y0=y[0];
       if(y0!=0){
        if(a0<ip){
          for(i=a0+1;i<=ip;i++)a[i]=0;
          for(i=1;i<=y0;i++)a[ip+i]=y[i];
          a[0]=ip+y0;
        }
        else{
          if(a0<(ip+y0)){
            for(i=a0+1;i<=(ip+y0);i++)a[i]=y[i-ip];
            tmp=a0;a0=ip+y0;y0=tmp-ip;
          }
```

```
        a[a0+1]=0;
        for(i=1;i<=y0;i++){
          a[ip+i]+=y[i];
          if(a[ip+i]>=BAZA){
            a[ip+i]-=BAZA;
            a[ip+i+1]+=1;
          }
        }
        i=ip+y0+1;
        while(a[i]>=BAZA){
          a[i]-=BAZA;i++;
          a[i]+=1;
        }
        if(a[a0+1]!=0)a0++;
        a[0]=a0;
      }
    }
  } /* dodaj */

  /* SRACUNAJ: Ostvaruje peti korak Schonhage Strassenovog algoritma. */
  void sracunaj(int yp,int ip,int k,int l,int *y){
    int nv,i,bz1,y0,j,spc;
    if(y[0]==-1){
      nv=sdva[l-SB];
      if(ip>yp){
        y[1]=yp;y[nv+1]=yp+1;y[0]=nv+1;
        for(i=2;i<=nv;i++)y[i]=0;
      }
      else{
        yp=sdva[k]-yp;y[1]=yp;
        if(yp==1)y[0]=-1;
        else{
          y[nv+1]=yp-1;y[0]=-(nv+1);
          for(i=2;i<=nv;i++)y[i]=0;
        }
      }
    }
    else{
      bz1=BAZA-1;
      if(yp!=0){
        nv=sdva[l-SB];
        if(yp>ip){
          y0=y[0];yp=sdva[k]-yp;
          if(y0==0){
            y[1]=y[nv+1]=yp;y[0]=-(nv+1);
            for(i=2;i<=nv;i++)y[i]=0;
          }
          else{
            y[1]=yp-y[1];
            if(y[1]<0){
              y[1]+=BAZA;
              for(i=2;i<=y0;i++)y[i]=bz1-y[i];
              for(i=y0+1;i<=nv;i++)y[i]=bz1;
              y[nv+1]=yp-1;
              if(yp==1)y[0]=-nv;
              else y[0]=-(nv+1);
            }
            else{
              if(y0>1){
                i=2;
                while(y[i]==0)i++;
                y[i]=BAZA-y[i];
                for(j=i+1;j<=y0;j++)y[j]=bz1-y[j];
```

```
                        for(i=y0+1;i<=nv;i++)y[i]=bz1;
                        y[nv+1]=yp-1;
                        if(yp==1)y[0]=-nv;
                        else y[0]=-(nv+1);
                      }
                      else{
                        for(i=2;i<=nv;i++)y[i]=0;
                        y[nv+1]=yp;y[0]=-(nv+1);
                      }
                    }
                  }
                                  }/* if(yp>ip) */
                else{
                  spc=0;y0=y[0];
                  if(y0==0){
                    y[1]=y[nv+1]=yp;
                    for(i=2;i<=nv;i++)y[i]=0;
                    y[0]=nv+1;
                  }
                  else{
                    if((yp==ip)&&(y0==nv)&&((y[1]+ip)>=BAZA)){
                      i=nv;
                      while(y[i]==bz1)i--;
                      if(i<2){
                        spc=1;yp=sdva[k]-yp;y[nv+1]=yp-1;
                        if(yp==1){
                          y[1]=yp-y[1];
                          if(y[1]<0){
                            y[1]+=BAZA;y[0]=-1;
                          }
                          else{
                            y[2]=1;y[0]=-2;
                          }
                        }
                        else{
                          for(i=3;i<=nv;i++)y[i]=0;
                          y[0]=-(nv+1);y[1]=yp-y[1];
                          if(y[1]<0){
                            y[1]+=BAZA;y[2]=0;
                          }
                          else{
                            y[2]=1;
                          }
                        }
                      }
                    }/* if((yp==ip)&&... */
                    if(spc==0){
                      y[nv+1]=y[y0+1]=0;
                      y[1]+=yp;i=1;
                      while(y[i]>=BAZA){
                        y[i]-=BAZA;i++;
                        y[i]+=1;
                      }
                      for(i=y0+2;i<=nv;i++)y[i]=0;
                      y[nv+1]+=yp;
                      y[0]=nv+1;
                    }
                  }
                }
              }
            }
          }
        }
      } /* sracunaj */
```

```
/* KVADRIRAJ: Vrsi kvadriranje datog broja po modulu F_n.
              Procedura se zasniva na Schonhage Strassenovom algoritmu.
*/
void kvadriraj(int *a,int n){
   int l,k,sfi,somg,a0,i,izdnl,dnl,izdnl1;
   static int u[DUZ_UI][DUZ_UJ],up[2*DUZ_UI],b[DUZ_A];
   int mb,bm,j,ip,u0;
   if(a[0]<33){
      square(a);moduo(a,n);
   }
   else{
      while(a[0]<=sdva[n-2-SB])n--;
      l=n/2;k=n-l;
      sfi=sdva[l+1-k];somg=2*sfi;
      a0=a[0];i=0;izdnl=dnl=sdva[l-SB];
      izdnl1=0;bm=sdva[k];mb=bm-1;
      while(izdnl<a0){
         u0=dnl;
         while((u0>0)&&(a[izdnl1+u0]==0))u0--;
         u[i][0]=u0;up[i]=a[izdnl1+1] & mb;
         for(j=1;j<=u0;j++)u[i][j]=a[izdnl1+j];
         i++;izdnl1=izdnl;izdnl+=dnl;
      }
      u0=a0-izdnl1;u[i][0]=u0;up[i]=a[izdnl1+1] & mb;
      for(j=1;j<=u0;j++)u[i][j]=a[izdnl1+j];
      for(j=i+1;j<bm;j++){
         u[j][0]=up[j]=0;
      }
          mult(up,k,mb);
      for(i=0;i<bm;i++)
         up[i]=((up[i] & mb)-(up[bm+i] & mb)+bm) & mb;
      for(i=1;i<bm;i++)pomeri(&u[i][0],i*sfi,l+1);
      fft(u,k,l+1,somg);
          for(i=0;i<bm;i++){
             square(&u[i][0]);
             moduo(&u[i][0],l+1);
      }
      ffti(u,k,l+1,-somg);
      for(i=0;i<bm;i++)pomeri(&u[i][0],-(i*sfi+k),l+1);
      for(i=0;i<bm;i++){
         if(u[i][0]>0)up[i]=(up[i]-(u[i][1]&mb)+bm)&mb;
      }
      for(i=0;i<bm;i++)sracunaj(up[i],i,k,l+1,&u[i][0]);
      a[0]=0;b[0]=0;
      for(i=0;i<bm;i++){
         ip=i*sdva[l-SB];
         if(u[i][0]<0){
            u[i][0]*=(-1);
            dodaj(b,&u[i][0],ip);
         }
         else{
            dodaj(a,&u[i][0],ip);
         }
      }
      moduo(a,n);moduo(b,n);
      modsub(a,b,a,n);
   }
}/*kvadriraj */

/* SMESTI: Upisuje u odgovarajucu datotcku dati broj. */
void smesti(int *a,FILE *fp,int dd){
   int i,j;
   i=0;
```

```
   fprintf(fp," %d %d\n",a[0],dd);
   while(i<a[0]){
      j=0;
      while((j<12) && (i<a[0])){
         j++;i++;
         fprintf(fp," %x",a[i]);
      }
      fprintf(fp,"\n");
   }
}/* smesti */

/* MAIN: Ostvaruje Pepinov test pozivajuci proceduru kvadriraj. */
main(){
   static int a[DUZ_A];
   int i,j,n,poc,kraj;
   char *imed;
   FILE *fp;
   inic();
   #if POJEDIN
         printf("\n Poco sam, unesi n= ");
         scanf("%d",&n);getchar();
         poc=n;kraj=n;
   #else
         printf("\n Pocco sam. Radim za n =5,6,..., %d \n",MAX_N);
         poc=5;kraj=MAX_N;
   #endif
   for(j=poc;j<=kraj;j++){
         a[0]=1;a[1]=3;n=j;
         for(i=1;i<sdva[n];i++){
            kvadriraj(a,n);
         }
         if(n>9){
            imed="tr10d.dat";imed[3]=38+n;
         }
         else{
            imed="tr00d.dat";imed[3]=48+n;
         }
         fp=fopen(imed,"w+");
         smesti(a,fp,0);
         fclose(fp);
         printf("\n n=%d,  t=%f ",n,clock()/CLK_TCK);
   }
   printf("\n KRAJ \n");getchar();
}/* main */
```