Osnovna svojstva DFT

1. DFT je linearna transformacija.
   To je očito iz vektorsko-matričnog zapisa
   $$y = V_n \cdot a \qquad \text{za} \quad y = DFT_n(a)$$

2. Pomaci i periodičnost.
   U "definicionim" relacijama
   $$y_k = A(\omega_n^k) = \sum_{j=0}^{n-1} a_j \omega_n^{k \cdot j}, \quad k = 0, \dots, n-1$$
   lijeva strana se može korektno definirati za bilo koji $k \in \mathbb{Z}$. Analogno vrijedi i za obratnu transformaciju:
   $$a_j = \frac{1}{n} \cdot \sum_{k=0}^{n-1} y_k \omega_n^{-k \cdot j}, \quad j = 0, \dots, n-1$$
   proširenjem na $j \in \mathbb{Z}$.
   Dobivamo 2 dvostrano beskonačna niza
   $$(y_k), \ (a_j)$$
   i znamo da su transf. inverzne na skupu indeksa
   $$\{0, \dots, n-1\}.$$

   Neka (y_{k-s}) označava niz sa pomaknutim članovima i analogno za (a_{j-s}).
   Tada je
   $$DFT_n(a_{j-s}) = (\omega_n^{ks} y_k), \quad \text{uz } (y_k) = DFT_n(a_j)$$
   i
   $$DFT_n^{-1}(y_{k-s}) = (\omega_n^{-js} a_j), \quad \text{uz } (a_j) = DFT_n^{-1}(y_k)$$

   Ako je s višekratnik od n, onda je
   $$\omega_n^{ks} = \omega_n^{-js} = [(\omega_n)^n]^{\text{nešto}} = 1, \quad \forall k, j$$
   pa dobivamo originalne nizove!
   To opravdava periodičko proširenje s periodom $\underline{\underline{n}}$.

3. Konvolucijski teorem:

$$DFT_n(a \otimes b) = DFT_n(a) \cdot DFT_n(b)$$

(što smo već koristili i, zapravo, dokazali, ali za posebne nizove a', b').

Diskretna Fourier-ova Transformacija – DFT

Zadan je $n \in \mathbb{N}$ i konačni niz ili vektor $a \in \mathbb{C}^n$ od n elemenata

$$a = (a_0, a_1, \ldots, a_{n-1})^T.$$

Taj niz/vektor možemo interpretirati i kao funkciju

$$a : \mathbb{Z}_n \to \mathbb{C}$$

gdje je $\mathbb{Z}_n = \{0, \ldots, n-1\}$ standardni sustav ostataka modulo n.

Elemente vektora a indeksiramo elementima iz $\mathbb{Z}_n$, a ne iz $\{1, \ldots, n\}$, kao što je uobičajeno u linearnoj algebri. Razlog nije samo tradicija u obradi signala, nego ima i matematičku pozadinu.

Vektor a možemo zamišljati kao diskretni uzorak nekog (može i kontinuiranog) signala, koji je "snimljen" u trenucima $t = 0, \ldots, n-1$.

Vektoru a možemo prirodno pridružiti polinom A nad $\mathbb{C}$, oblika

$$A(x) = \sum_{j=0}^{n-1} a_j x^j .$$

Takvi polinomi (kao i vektori a, duljine n) čine vektorski prostor nad $\mathbb{C}$, dimenzije n, izomorfan s $\mathbb{C}^n$. Zato kažemo da polinom A ima <u>red n</u>, tj. stupanj je $\leq n-1$.

Vektore, gledane kao funkcije na $\mathbb{Z}_n$, možemo i množiti – po točkama. Analogno, polinome možemo množiti (algebra!). Međutim, odmah uočavamo da ovakvo pridruživanje

$$a \longmapsto A$$

<u>nije</u> homomorfizam obzirom na množenje (Očito je homomorfizam na ostale operacije – zbrajanje i množenje skalarom – tj. između vektorskih prostora).

Naime, za "funkcijsko" ili Hadamard-ovo množenje vektora vrijedi

$$c = a \cdot b \iff c_j = a_j \cdot b_j, \; j = 0, \ldots, n-1,$$

dok za množenje polinoma vrijedi

$$C = A \cdot B \iff c_j = \sum_{k=0}^{j} a_k \cdot b_{j-k}, \; j = 0, \ldots, \underline{2n-2}$$

pa je produkt C reda $2n-1$.

Da bismo uspostavili vezu s produktom polinoma, na vektorima uvodimo novu operaciju $\otimes$, koju zovemo <u>konvolucija</u>, tako da je

$$(a \otimes b)_j = \sum_{k=0}^{n-1} a_k \, b_{j-k}, \; j = 0, \ldots, n-1.$$

↳ periodičko pros.

Ovo još uvijek ne odgovara relaciji za koeficijente produkta polinoma. I ne može, jer redovi (duljine) nisu ujednačeni.

Međutim, ako nizove/vektore a i b produljimo nulama do dovoljne duljine N, pripadni polinomi A i B ostaju <u>isti</u>, samo ih gledamo u većem vektorskom prostoru. Naravno, i $C = A \cdot B$ se ne mijenja. Zbog toga je dovoljno uzeti $N \geq 2n-1$, pa da i niz c koeficijenata produkta bude <u>cijeli</u> prikaziv.

Dakle, napravimo

$$\left.\begin{array}{l} a \mapsto a' = (a_0, \ldots, a_{n-1}, 0, \ldots, 0)^T \\ b \mapsto b' = (b_0, \ldots, b_{n-1}, 0, \ldots, 0)^T \end{array}\right\} \text{duljine } N$$

i $\quad c' = a' \otimes b' = (c'_0, \ldots, c'_{N-1})$

Ako je $N \geq 2n-1$, onda vrijedi

$$c'_j = \sum_{k=0}^{N-1} a'_k \, b'_{j-k} = \sum_{k=0}^{j} a_k \, b_{j-k}, \; j = 0, \ldots, 2n-2$$

$$c'_j = 0, \; j > 2n-2 \quad (j = 2n-1, \ldots, N-1)$$

i stavimo:

$$\underline{c := (c'_0, \ldots, c'_{2n-2})^T} \text{ - duljine } 2n-1.$$

Dobiveni niz c je pridružen polinomu $C = A \cdot B$.
(Preciznije, dobivenom nizu c pridružen je polinom $C = A \cdot B$).

Ovime smo uspostavili formalni "homomorfizam" konvolucije nizova i produkta polinoma. Stvarno, još ništa korisno nismo napravili, dok nemamo efikasne algoritme za računanje nečeg od toga!

Polinome zasad prikazujemo koeficijentima — u tzv. koeficijentnoj reprezentaciji. Ako se opet sjetimo interpolacije, možemo koristiti i drugačiji prikaz — vrijednostima na dovoljno velikom skupu točaka.

Sasvim općenito, za polinom A reda n možemo odabrati bilo koji skup od točno n različitih točaka $z_0, \ldots, z_{n-1} \in \mathbb{C}$ i dobiti tzv. vrijednosnu reprezentaciju — vektorom vrijednosti

$$\left(A(z_0), \ldots, A(z_{n-1}) \right)^T \in \mathbb{C}^n .$$

Uočimo da u ovakvoj reprezentaciji, sve aritmetičke operacije na polinomima imaju jednostavan oblik — po točkama. To vrijedi i za produkt!

$$(A \cdot B)(z) = A(z) \cdot B(z)$$

Ako pazimo na red, sve operacije su efikasne — brzo se izvode — linearno u duljini nizova.

Ono što nam fali je efikasan prijelaz između te dvije reprezentacije

vektor koef. $\longleftrightarrow$ vektor vrijednosti.

Napomena: ovo ima smisla samo za produkt, ako nam uspije.
Zašto? Sve ostale operacije (zbrajanje, množenje skalarom) idu brzo u obje reprezentacije — tj. i u koeficijentnoj. Međutim, za množenje imamo samo standardni $O(n^2)$ algoritam u koeficijentnoj reprezentaciji, pa tu tražimo uštedu!

Efikasni prijelaz iz jedne u drugu reprezentaciju (i natrag) dobivamo pametnim izborom točaka $z_0, \dots, z_{n-1}$.

Diskretna Fourierova Transformacija - DFT

Želimo izračunati vrijednost polinoma A, reda n (stupnja $n-1$)

$$A(x) = \sum_{j=0}^{n-1} a_j x^j$$

u svim n-tim korijenima iz jedinice, tj. u točkama

$$\omega_n^0, \omega_n^1, \dots, \omega_n^{n-1} \qquad \left(\begin{array}{l} z_k = \omega_n^k \\ k = 0, \dots, n-1 \end{array} \right)$$

gdje je

$$\omega_n = e^{2\pi i/n}.$$

Kasnije ćemo uzeti da je n potencija od 2, $n = 2^m$, ali trenutno sve vrijedi za bilo koji n.

Tj. treba izračunati vektor $y = (y_0, \dots, y_{n-1})^T$ vrijednosti

$$y_k = A(\omega_n^k) = \sum_{j=0}^{n-1} a_j \omega_n^{k \cdot j}, \quad k = 0, \dots, n-1.$$

Vektor vrijednosti $y = (y_0, \dots, y_{n-1})^T$ zovemo diskretna Fourierova transformacija vektora koeficijenata $a = (a_0, \dots, a_{n-1})^T$. Zapis je

$$y = DFT_n(a)$$

$n \to$ označava duljinu (dimenziju) uzorka – vektora.

Ovu operaciju možemo pisati i u matričnom obliku

$$y = V_n \cdot a$$

gdje V_n matrica reda n, s elementima

$$(V_n)_{kj} = \omega_n^{kj}, \quad j, k = 0, \dots, n-1$$

$\hookrightarrow$ nije baš uobičajeno da su indeksi iz $\{0, \dots, n-1\}$, već iz $\{1, \dots, n\}$, ali je ovdje puno zgodnije.

Uočimo odmah da je V_n specijalna Vandermondeova matrica ($x_j = \omega_n^j$).

Vidimo odmah da je V_n simetrična (kompleksna) matrica

$$V_n = V_n^T$$

(ali nije Hermitska).

Njezin inverz se lako računa iz svojstava n-tih korijena iz jedinice.

Za $n \in \mathbb{N}$ i $k \in \mathbb{N}_0$, ako n ne dijeli k, onda je

$$\sum_{j=0}^{n-1} (\omega_n^k)^j = 0 \qquad = \left|\frac{k}{n}\right|, \text{ ako } n \text{ dijeli } k \ (k \in \mathbb{Z}).$$

Koristeći ovo, lako se dokazuje da za inverz V_n^{-1} vrijedi

$$V_n^{-1} = \frac{1}{n} \cdot V_n^*$$

(tj. $U_n := \frac{1}{\sqrt{n}} V_n$
$U_n^{-1} = \sqrt{n} \cdot V_n^{-1} = \frac{\sqrt{n}}{n} \cdot V_n^*$
$= \frac{1}{\sqrt{n}} V_n^* = (U_n)^*$
$\Rightarrow U_n$ ortogonalna ili, preciznije, unitarna!)

Posebno,

$$\overline{\omega_n} = \omega_n^{-1}.$$

Po elementima je:

$$(V_n^{-1})_{kj} = \frac{1}{n} \omega_n^{-kj}.$$

Inverzna diskretna Fourierova transformacija je onda

$$a = DFT_n^{-1}(y)$$

ili

$$a = V_n^{-1} y$$

tj. ako DFT ide s ω_n onda DFT^{-1} ide s ω_n^{-1} i još, na kraju, pomnoži sve $\frac{1}{n}$

odnosno, po elementima

$$a_j = \frac{1}{n} \cdot \sum_{k=0}^{n-1} y_k \, \omega_n^{-k \cdot j}, \quad j = 0, \dots, n-1.$$

Nap: – Kad se drukčije definiraju DFT i DFT^{-1}
– periodičnost $\omega_n^k = \omega_n^{k \bmod n}, \ \forall k \in \mathbb{Z}$.
– grupa n-tih korijena iz jed. (mult.) $\approx (\mathbb{Z}_n, +_n)$
– halving lemma [n paran, $\omega_n^2 = \omega_{n/2}$] ⟸ $\omega_{dn}^{dk} = \omega_n^k$, $n \geq 0$, $k \geq 0$, $d > 0$
– koef. op's = elementary!

$\omega_n^{n/2} = -1$, n par

$(\omega_n^{k+\frac{n}{2}})^2 = (\omega_n^k)^2$, n par

DFT ima još jedno važno svojstvo, vezano uz produkte.

Definiramo još dvije operacije $\cdot$ i $\otimes$, za $a, b \in \mathbb{C}^n$

$\cdot$ = Hadamardov produkt – produkt po točkama (elementima)

$$a \cdot b = (a_0 \cdot b_0, \ldots, a_{n-1} \cdot b_{n-1})^T \qquad (a \cdot b)_j = a_j b_j$$
$$j = 0, \ldots, n-1$$

$\otimes$ = konvolucija

$$c = a \otimes b, \text{ uz } \quad c_j = \sum_{k=0}^{j} a_k b_{j-k}, \; j = 0, \ldots, n-1.$$

Praktična korist u kontekstu množenja polinoma: ako su a, b vektori koeficijenata polinoma $A(x), B(x)$ onda je $c = a \otimes b$ vektor koeficijenata njihovog produkta $C(x) = A(x) \cdot B(x)$.

Tu treba biti malo oprezan, zbog duljine vektora, jer produkt ima dvostruki stupanj (odn. sumu stupnjeva)

$$\deg C = \deg A + \deg B$$

Zbog toga, vektore a i b treba dopuniti do duljine od vektora c, i to – očito – nulama.

$$A(x) = \sum_{j=0}^{n-1} a_j x^j, \quad B(x) = \sum_{j=0}^{n-1} b_j x^j, \quad C(x) = \sum_{j=0}^{2n-2} c_j x^j$$

($c_{2n-1} = 0 \Rightarrow$ dobijem dvostruku duljinu)

Općenito

$$a \otimes b = DFT_n^{-1}(DFT_n(a) \cdot DFT_n(b))$$
$$(2n) \qquad (2n) \qquad (2n)$$

$a \mapsto a'$
$b \mapsto b'$

$$a' \otimes b' = DFT_{2n}^{-1}(DFT_{2n}(a') \cdot DFT_{2n}(b'))$$

$n = 2^m$ za $m = \varnothing$ $T(1) = \varnothing$

općenito at stage $n = 2^m$ – imam for od $\varnothing$ do $n/2 - 1$
tj. 2^{m-1} × :

$$\left.\begin{matrix} 1A + 1M \\ 1A + 1M \end{matrix}\right\} \quad 1M + 2A$$

$$1M \text{ ——— } \underline{1M}$$

$$2M + 2A$$

ili $2^m \cdot M + 2^m \cdot A$

Dakle:
$$M(n) = 2 \cdot M(n/2) + n \quad \text{i isto za } A$$
$$M(n/2) = 2 \cdot M(n/4) + n/2$$

$$\Rightarrow M(n) = 4 \cdot M(n/4) + n + n$$
$$= 4 \cdot \left(2M\left(\tfrac{n}{8}\right) + \tfrac{n}{4}\right) + n + n$$
$$= 8 \cdot M\left(\tfrac{n}{8}\right) + 3 \cdot n$$

Općenito:
$$M(n) = 2^k \cdot M\left(\tfrac{n}{2^k}\right) + k \cdot n$$
$$= 2^k \cdot \left(2 \cdot M\left(\tfrac{n}{2^{k+1}}\right) + \tfrac{n}{2^k}\right) + k \cdot n$$
$$= 2^{k+1} M\left(\tfrac{n}{2^{k+1}}\right) + (k+1) \cdot n$$

za $k = \lg n$ $\quad M(n) = n \cdot \underbrace{M(1)}_{0} + n \cdot \lg n = n \cdot m$

Dakle, broj kompl. zbrajanja $\boxed{A(n) = n \cdot \lg n}$

– Ako spremim pomoćno: $\omega \cdot y_k^{[1]} \rightarrow \boxed{M_1(n) = \frac{1}{2} n \cdot \lg n}$

a oper. $\underbrace{\omega := \omega \cdot \omega_n}$ košta još $M_2(n) = \frac{1}{2} n \cdot \lg n$

ovo mogu tabelirati i spremiti
u polje duljine n za $M_2(n) = n - 1$

– Par FFT, FFT^{-1} → treba $2\times$ i dodati (barem) nM
za final scaling
(ili $2nM$, ako oba imaju const
na pr. $\frac{1}{\sqrt{n}}$)

Prošli put smo napravili diskretnu Fourierovu transformaciju DFT_n (periodičkog) niza duljine (perioda) n - u kontekstu brzog množenja polinoma nad $\mathbb{C}$ i napravili smo brzu rekurzivnu realizaciju FFT_n u slučaju da je $n = 2^m$, $m \in \mathbb{N}_0$ potencija od 2.

Cilj danas: - brza iterativna implementacija za FFT_n, $n = 2^m$, iz koje dobivamo i brzu paralelnu implementaciju
- proširenje na bilo koji $n \in \mathbb{N}$

Kasnije - razne primjene FFT:
- analiza vremenskih signala, izglađivanje i filtriranje (samo demo)
- konstrukcija brzih algoritama za razne probleme.

- Ponovimo ukratko definiciju DFT_n, za $n \in \mathbb{N}$.

Neka je $a \in \mathbb{C}^n$, $a = (a_0, \dots, a_{n-1})^T$. Diskretna Fourierova transformacija DFT_n je preslikavanje

$$DFT_n : \mathbb{C}^n \to \mathbb{C}^n$$

za koje je

$$y = DFT_n(a)$$

ako i samo ako je

$$y = V_n \cdot a$$

gdje je $V_n \in \mathbb{C}^{n \times n}$ matrica reda n, oblika

$$(V_n)_{kj} = \omega_n^{k \cdot j}, \quad j, k \in \{0, \dots, n-1\}$$

s tim da je

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

osnovni n-ti korijen iz jedinice.

Inverzna diskretna Fourierova transformacija DFT_n^{-1} je naravno

$$a = DFT_n^{-1}(y) \iff a = V_n^{-1} \cdot y.$$

Prošli puta smo vidjeli da je $V_n = V_n^T$ i da za inverz vrijedi

$$V_n^{-1} = \frac{1}{n} \cdot V_n^*.$$

Zbog $\overline{\omega}_n = \omega_n^{-1}$, DFT_n^{-1} dobivamo tako da u DFT_n, umjesto ω_n, koristimo $\omega_n^{-1} = \overline{\omega}_n$ i, na kraju skaliramo finalni vektor s $1/n$.

U algoritamskim implementacijama se vrlo često ignorira ovo skaliranje na kraju i dodaje samo kad je nužno potrebno i to na samom kraju svih transformacija.

Zbog toga, to skaliranje nećemo posebno brojati u analizi složenosti, opet, osim u kompliciranijim algoritmima – s više DFT, DFT^{-1} transformacija, kad je to skaliranje bitno za korektan konačni rezultat.

– Nekoliko komentara, zbog raznih oznaka i imena u literaturi.

1. U teoriji je najugodnije raditi s matricom

$$U_n = \frac{1}{\sqrt{n}} \cdot V_n$$

(simetrična skala za DFT_n i DFT_n^{-1} – oba imaju <u>istu</u> skalu), zato što je

$$U_n^{-1} = U_n^*,$$

pa je U_n unitarna, što je zgodno za razne stvari u teoriji (čuva skalarne produkte i norme vektora!).

Tada se DFT_n^{-1} dobiva iz DFT_n samo zamjenom

$$\omega_n \mapsto \omega_n^{-1} = \overline{\omega}_n$$

$$(DFT_n) \qquad (DFT_n^{-1}).$$

U tom smislu, mogli bismo za DFT_n i DFT_n^{-1} uzeti bilo koji par matrica

$$W_n = c_1 \cdot V_n \,, \quad c_2 \cdot V_n^* = W_n^{-1}$$

za koji vrijedi $\quad c_1 \cdot c_2 = \frac{1}{n}$.

No, najzgodnije je uzeti:

(a) iste skale, $c_1 = c_2 = \frac{1}{\sqrt{n}}$, što vodi na unitarne matrice

(b) jedna od te dvije konstante je jednaka 1 ($c_1 = 1$, $c_2 = \frac{1}{n}$ ili obratno), jer to štedi jedno skaliranje vektora (u DFT_n ili DFT_n^{-1}).

2. Vrlo često se susreće obratna definicija sa zamjenom uloga DFT_n i DFT_n^{-1}, tj.

DFT_n odgovara ω_n^{-1} s nekom skalom

DFT_n^{-1} odgovara ω_n — " —

(tako da je produkt skala opet $\frac{1}{n}$).

Tada se uobičajeno koristi i oznaka $\mathcal{F}$ za Fourierovu transformaciju, jer negativni eksponenti odgovaraju tzv. neprekidnoj Fourierovoj transformaciji

$$F = \mathcal{F}(f) \qquad F(x) = \int_{-\infty}^{\infty} f(t)\, e^{-2\pi i \cdot t \cdot x}\, dt$$

↓ a inverzna transformacija ima + u eksponentu.

Drugu analogiju ovog oblika dobivamo gledanjem Fourierovog reda ili razvoja periodičke funkcije f na $[0, 1]$ (recimo!). Fourierov red ima oblik

$$f(x) = \sum_{j=-\infty}^{\infty} c_j \cdot e^{2\pi i \cdot jx}$$

a za koeficijente vrijedi

$$c_j = \int_0^1 f(x) \cdot e^{-2\pi i \cdot jx}\, dx \,.$$

U našem putu do DFT_n za brzo množenje polinoma, prirodnije je bilo uzeti ω_n (tj. $z_k = \omega_n^k$), nego inverze ($z_k = \omega_n^{-k}$), zato je jasno da smo mogli raditi s negativnim potencijama ($z_k = (\omega_n^{-1})^k$).

$$y_k = A(\omega_n^k)\ , \quad \text{za} \quad A(x) = \sum_{j=0}^{n-1} a_j x^j\ , \quad k = 0, \ldots, n-1.$$

3. Kod nas je DFT_n linearni operator na $\mathbb{C}^n$, tj. djeluje na obične vektore $a \in \mathbb{C}^n$.

Zbog svojstava periodičnosti koja smo naveli prošli put, katkad se DFT_n odmah definira na periodičnim nizovima s periodom n.
(v. Henrici, Applied and Computational Complex Analysis, 1, 2 i 3, Wiley, 1986. – za Volume 3).

Kako izgleda takva konstrukcija odn. matematički kontekst?

Promatramo obostrano beskonačne nizove $x \in \mathbb{C}^{\mathbb{Z}}$

$$x = \{x_k\}_{k=-\infty}^{\infty} = \{x_k\}_{k \in \mathbb{Z}}.$$

Za zadani $n \in \mathbb{N}$, definiramo prostor Π_n svih takvih nizova s periodom n, tj. onih x za koje vrijedi

$$x_{k+n} = x_k, \quad \forall k \in \mathbb{Z}.$$

Bilo koji element $x \in \Pi_n$ je, naravno, potpuno određen bilo kojim blokom koji pokriva period, tj. bilo kojim blokom od n uzastopnih elemenata, na primjer, $x_0, \ldots, x_{n-1}$.

Oznaka za obostrano beskonačno periodičko proširenje je

$$x := \|: x_0, \ldots, x_{n-1} :\| \qquad (\text{ili } (: \ , :))$$

da ne bude zabune s normama.

Očito je Π_n vektorski prostor izomorfan s $\mathbb{C}^n$!
Dalje je jednostavno – svi objekti na Π_n, poput skalarnog produkta, norme, ... definiraju se na vektorima koji razapinju period $(x_0, \ldots, x_{n-1})$.

Brzi rekurzivni algoritam FFT za DFT_n, ako je $n = 2^m$, $m \in \mathbb{N}_0$, tj. n je potencija od 2, dobivamo rastavom polinoma A

$$A(x) = a_0 + a_1 x + a_2 x^2 + a_3 x^3 + \dots + a_{n-1} x^{n-1}$$

na parne i neparne koeficijente:

$$A(x) = (a_0 + a_2 x^2 + \dots + a_{n-2} x^{n-2}) + x \cdot (a_1 + a_3 x^2 + \dots + a_{n-1} x^{n-2})$$

i supstitucijom $x' = x^2$.

Dakle, definiramo "parni" i "neparni" polinom

$$A^{[0]}(x) = a_0 + a_2 x + \dots + a_{n-2} x^{n/2-1}$$
$$A^{[1]}(x) = a_1 + a_3 x + \dots + a_{n-1} x^{n/2-1}$$

pa je

$$A(x) = A^{[0]}(x^2) + x \cdot A^{[1]}(x^2).$$

Označimo s $a^{[0]}$ i $a^{[1]}$ pripadne nizove koeficijenata

$$a^{[0]} = (a_0, a_2, -, a_{n-2})$$
$$a^{[1]} = (a_1, a_3, -, a_{n-1}).$$

Vidimo da $a^{[0]}$ sadrži one koeficijente čiji indeks ima zadnju znamenku jednaku 0 u bazi 2, a $a^{[1]}$ one čiji indeks ima zadnju znamenku 1 u bazi 2.

Broj ili baza 2 je faktor cijepanja ili rastava

$$n = 2 \cdot \left(\frac{n}{2}\right)$$

stari red faktor · novi red.

Neka su $y^{[0]}$ i $y^{[1]}$ pripadne diskretne Fourierove transformacije polovnog reda

$$y^{[0]} := DFT_{n/2}(a^{[0]})$$
$$y^{[1]} := DFT_{n/2}(a^{[1]}).$$

"Puni" DFT polaznog niza koeficijenata

$$y = DFT_n(a)$$

dobivamo pažljivom kombinacijom "polovica" vektora $y^{[0]}$ i $y^{[1]}$

prva polovica: $y_k = y_k^{[\emptyset]} + \omega_n^k \cdot y_k^{[1]}$

druga polovica:

$$y_{k+n/2} = y_k^{[\emptyset]} + \omega_n^{k+n/2} \cdot y_k^{[1]}$$
$$= y_k^{[\emptyset]} - \omega_n^k \cdot y_k^{[1]}$$

za $k = 0, 1, \ldots, n/2$.

Vidimo da je drugi član isti, do na predznak, pa odmah možemo uštedjeti jedno množenje, ako izračunamo pomoćnu vrijednost

$$t = \omega_n^k \cdot y_k^{[1]}$$

a zatim

$$y_k = y_k^{[\emptyset]} + t, \quad y_{k+n/2} = y_k^{[\emptyset]} - t.$$

Odgovarajući rekurzivni algoritam, pisan funkcijski (nalik na Matlab ili C) ima oblik:

```
function FFT (a);   { a je kompleksni vektor,
                      vraća y = DFT_n(a) }
  n := length(a);   { pretpostavka je n = 2^m, m ∈ N_0 ! }
  if n = 1 then
    return a        { y = a za n = 1 }
  else
    a^[∅] := (a_0, a_2, ..., a_{n-2});
    a^[1] := (a_1, a_3, ..., a_{n-1});
    y^[∅] := FFT(a^[∅]);
    y^[1] := FFT(a^[1]);
    ω_n := e^{2πi/n}; { ω := 1; }
    for k := ∅ to n/2-1 do          → n/2-1
      t := ω_n^k · y_k^[1];   { = ω · y_k^[1] }
      y_k := y_k^[∅] + t;
      y_{k+n/2} := y_k^[0] - t;
      { ω := ω * ω_n }
    end for ;
  return y;   { kompleksni vektor duljine n }
```

Skaliranje, ako treba, napravimo nakon povratka!

[Ovdje dobije složenost $\left\{ \begin{matrix} \frac{1}{2} n \lg n \, M \\ n \lg n \, A \end{matrix} \right\}$ uz table-lookup za ω_n^k !]

since the degree-bound of C is $2n$, Theorem 32.1 implies that we need $2n$ point-value pairs for a point-value representation of C. We must therefore begin with "extended" point-value representations for A and for B consisting of $2n$ point-value pairs each. Given an extended point-value representation for A,

$$\{(x_0, y_0), (x_1, y_1), \ldots, (x_{2n-1}, y_{2n-1})\} ,$$

and a corresponding extended point-value representation for B,

$$\{(x_0, y'_0), (x_1, y'_1), \ldots, (x_{2n-1}, y'_{2n-1})\} ,$$

then a point-value representation for C is

$$\{(x_0, y_0 y'_0), (x_1, y_1 y'_1), \ldots, (x_{2n-1}, y_{2n-1} y'_{2n-1})\} .$$

Given two input polynomials in extended point-value form, we see that the time to multiply them to obtain the point-value form of the result is $\Theta(n)$, much less than the time required to multiply polynomials in coefficient form.

Finally, we consider how to evaluate a polynomial given in point-value form at a new point. For this problem, there is apparently no approach that is simpler than converting the polynomial to coefficient form first, and then evaluating it at the new point.

Fast multiplication of polynomials in coefficient form

Can we use the linear-time multiplication method for polynomials in point-value form to expedite polynomial multiplication in coefficient form? The answer hinges on our ability to convert a polynomial quickly from coefficient form to point-value form (evaluate) and vice-versa (interpolate).

We can use any points we want as evaluation points, but by choosing the evaluation points carefully, we can convert between representations in only $\Theta(n \lg n)$ time. As we shall see in Section 32.2, if we choose "complex roots of unity" as the evaluation points, we can produce a point-value representation by taking the Discrete Fourier Transform (or DFT) of a coefficient vector. The inverse operation, interpolation, can be performed by taking the "inverse DFT" of point-value pairs, yielding a coefficient vector. Section 32.2 will show how the FFT performs the DFT and inverse DFT operations in $\Theta(n \lg n)$ time.

Figure 32.1 shows this strategy graphically. One minor detail concerns degree-bounds. The product of two polynomials of degree-bound n is a polynomial of degree-bound $2n$. Before evaluating the input polynomials A and B, therefore, we first double their degree-bounds to $2n$ by adding n high-order coefficients of 0. Because the vectors have $2n$ elements, we use "complex $(2n)$th roots of unity," which are denoted by the ω_{2n} terms in Figure 32.1.

Given the FFT, we have the following $\Theta(n \lg n)$-time procedure for multiplying two polynomials $A(x)$ and $B(x)$ of degree-bound n, where the

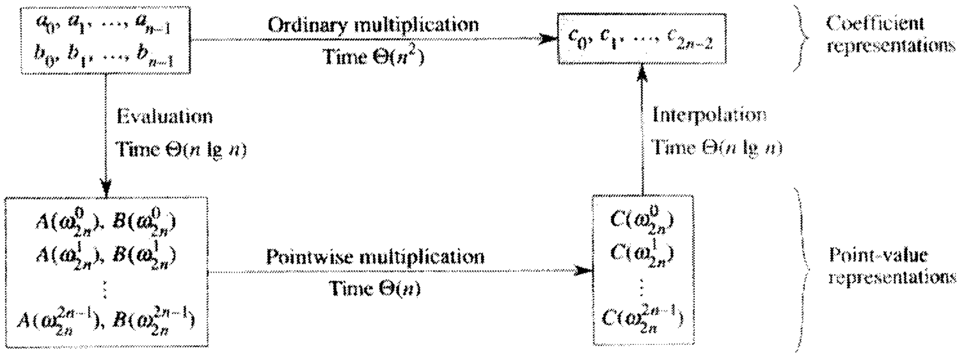


Figure 32.1 A graphical outline of an efficient polynomial-multiplication process. Representations on the top are in coefficient form, while those on the bottom are in point-value form. The arrows from left to right correspond to the multiplication operation. The ω_{2n} terms are complex $(2n)$th roots of unity.

input and output representations are in coefficient form. We assume that n is a power of 2; this requirement can always be met by adding high-order zero coefficients.

1. *Double degree-bound:* Create coefficient representations of $A(x)$ and $B(x)$ as degree-bound $2n$ polynomials by adding n high-order zero coefficients to each.
2. *Evaluate:* Compute point-value representations of $A(x)$ and $B(x)$ of length $2n$ through two applications of the FFT of order $2n$. These representations contain the values of the two polynomials at the $(2n)$th roots of unity.
3. *Pointwise multiply:* Compute a point-value representation for the polynomial $C(x) = A(x)B(x)$ by multiplying these values together pointwise. This representation contains the value of $C(x)$ at each $(2n)$th root of unity.
4. *Interpolate:* Create the coefficient representation of the polynomial $C(x)$ through a single application of an FFT on $2n$ point-value pairs to compute the inverse DFT.

Steps (1) and (3) take time $\Theta(n)$, and steps (2) and (4) take time $\Theta(n \lg n)$. Thus, once we show how to use the FFT, we will have proven the following.

Theorem 32.2

The product of two polynomials of degree-bound n can be computed in time $\Theta(n \lg n)$, with both the input and output representations in coefficient form. ∎

Exercises

32.1-1
Multiply the polynomials $A(x) = 7x^3 - x^2 + x - 10$ and $B(x) = 8x^3 - 6x + 3$ using equations (32.1) and (32.2).

32.1-2
Evaluating a polynomial $A(x)$ of degree-bound n at a given point x_0 can also be done by dividing $A(x)$ by the polynomial $(x - x_0)$ to obtain a quotient polynomial $q(x)$ of degree-bound $n - 1$ and a remainder r, such that

$$A(x) = q(x)(x - x_0) + r .$$

Clearly, $A(x_0) = r$. Show how to compute the remainder r and the coefficients of $q(x)$ in time $\Theta(n)$ from x_0 and the coefficients of A.

32.1-3
Derive a point-value representation for $A^{\text{rev}}(x) = \sum_{j=0}^{n-1} a_{n-1-j}x^j$ from a point-value representation for $A(x) = \sum_{j=0}^{n-1} a_j x^j$, assuming that none of the points is 0.

32.1-4
Show how to use equation (32.5) to interpolate in time $\Theta(n^2)$. (*Hint:* First compute $\prod_j(x - x_k)$ and $\prod_j(x_j - x_k)$ and then divide by $(x - x_k)$ and $(x_j - x_k)$ as necessary for each term. See Exercise 32.1-2.)

32.1-5
Explain what is wrong with the "obvious" approach to polynomial division using a point-value representation. Discuss separately the case in which the division comes out exactly and the case in which it doesn't.

32.1-6
Consider two sets A and B, each having n integers in the range from 0 to $10n$. We wish to compute the ***Cartesian sum*** of A and B, defined by

$$C = \{x + y : x \in A \text{ and } y \in B\} .$$

Note that the integers in C are in the range from 0 to $20n$. We want to find the elements of C and the number of times each element of C is realized as a sum of elements in A and B. Show that the problem can be solved in $O(n \lg n)$ time. (*Hint:* Represent A and B as polynomials of degree $10n$.)

32.2 The DFT and FFT

In Section 32.1, we claimed that if we use complex roots of unity, we can evaluate and interpolate in $\Theta(n \lg n)$ time. In this section, we define

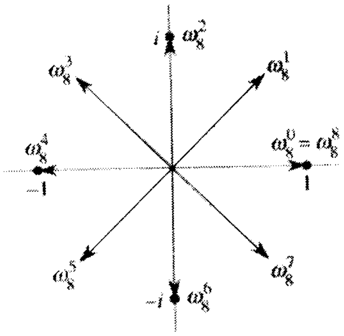


Figure 32.2 The values of $\omega_8^0, \omega_8^1, \ldots, \omega_8^7$ in the complex plane, where $\omega_8 = e^{2\pi i/8}$ is the principal 8th root of unity.

complex roots of unity and study their properties, define the DFT, and then show how the FFT computes the DFT and its inverse in just $\Theta(n \lg n)$ time.

Complex roots of unity

A ***complex nth root of unity*** is a complex number ω such that

$$\omega^n = 1 .$$

There are exactly n complex nth roots of unity: these are $e^{2\pi ik/n}$ for $k = 0, 1, \ldots, n-1$. To interpret this formula, we use the definition of the exponential of a complex number:

$$e^{iu} = \cos(u) + i\sin(u) .$$

Figure 32.2 shows that the n complex roots of unity are equally spaced around the circle of unit radius centered at the origin of the complex plane. The value

$$\omega_n = e^{2\pi i/n} \tag{32.6}$$

is called ***the principal nth root of unity***; all of the other complex nth roots of unity are powers of ω_n.

The n complex nth roots of unity,

$$\omega_n^0, \omega_n^1, \ldots, \omega_n^{n-1} ,$$

form a group under multiplication (see Section 33.3). This group has the same structure as the additive group $(\mathbf{Z}_n, +)$ modulo n, since $\omega_n^n = \omega_n^0 = 1$ implies that $\omega_n^j \omega_n^k = \omega_n^{j+k} = \omega_n^{(j+k) \bmod n}$. Similarly, $\omega_n^{-1} = \omega_n^{n-1}$. Essential properties of the complex nth roots of unity are given in the following lemmas.

Lemma 32.3 (Cancellation lemma)
For any integers $n \geq 0$, $k \geq 0$, and $d > 0$,

$$\omega_{dn}^{dk} = \omega_n^k . \tag{32.7}$$

Proof The lemma follows directly from equation (32.6), since

$$\begin{aligned} \omega_{dn}^{dk} &= \left(e^{2\pi i/dn}\right)^{dk} \\ &= \left(e^{2\pi i/n}\right)^{k} \\ &= \omega_n^k . \end{aligned}$$

■

Corollary 32.4
For any even integer $n > 0$,

$$\omega_n^{n/2} = \omega_2 = -1 .$$

Proof The proof is left as Exercise 32.2-1. ■

Lemma 32.5 (Halving lemma)
If $n > 0$ is even, then the squares of the n complex nth roots of unity are the $n/2$ complex $(n/2)$th roots of unity.

Proof By the cancellation lemma, we have $(\omega_n^k)^2 = \omega_{n/2}^k$, for any nonnegative integer k. Note that if we square all of the complex nth roots of unity, then each $(n/2)$th root of unity is obtained exactly twice, since

$$\begin{aligned} (\omega_n^{k+n/2})^2 &= \omega_n^{2k+n} \\ &= \omega_n^{2k}\omega_n^n \\ &= \omega_n^{2k} \\ &= (\omega_n^k)^2 . \end{aligned}$$

Thus, ω_n^k and $\omega_n^{k+n/2}$ have the same square. This property can also be proved using Corollary 32.4, since $\omega_n^{n/2} = -1$ implies $\omega_n^{k+n/2} = -\omega_n^k$, and thus $(\omega_n^{k+n/2})^2 = (\omega_n^k)^2$. ■

As we shall see, the halving lemma is essential to our divide-and-conquer approach for converting between coefficient and point-value representations of polynomials, since it guarantees that the recursive subproblems are only half as large.

Lemma 32.6 (Summation lemma)
For any integer $n \geq 1$ and nonnegative integer k not divisible by n,

$$\sum_{j=0}^{n-1} (\omega_n^k)^j = 0 .$$

Proof Because equation (3.3) applies to complex values,

$$\begin{aligned}\sum_{j=0}^{n-1} (\omega_n^k)^j &= \frac{(\omega_n^k)^n - 1}{\omega_n^k - 1} \\ &= \frac{(\omega_n^n)^k - 1}{\omega_n^k - 1} \\ &= \frac{(1)^k - 1}{\omega_n^k - 1} \\ &= 0 .\end{aligned}$$

Requiring that k not be divisible by n ensures that the denominator is not 0, since $\omega_n^k = 1$ only when k is divisible by n. ∎

The DFT

Recall that we wish to evaluate a polynomial

$$A(x) = \sum_{j=0}^{n-1} a_j x^j$$

of degree-bound n at $\omega_n^0, \omega_n^1, \omega_n^2, \ldots, \omega_n^{n-1}$ (that is, at the n complex nth roots of unity).[2] Without loss of generality, we assume that n is a power of 2, since a given degree-bound can always be raised—we can always add new high-order zero coefficients as necessary. We assume that A is given in coefficient form: $a = (a_0, a_1, \ldots, a_{n-1})$. Let us define the results y_k, for $k = 0, 1, \ldots, n-1$, by

$$\begin{aligned} y_k &= A(\omega_n^k) \\ &= \sum_{j=0}^{n-1} a_j \omega_n^{kj} . \end{aligned} \tag{32.8}$$

The vector $y = (y_0, y_1, \ldots, y_{n-1})$ is the ***Discrete Fourier Transform (DFT)*** of the coefficient vector $a = (a_0, a_1, \ldots, a_{n-1})$. We also write $y = \text{DFT}_n(a)$.

---

[2]The length n is actually what we referred to as $2n$ in Section 32.1, since we double the degree-bound of the given polynomials prior to evaluation. In the context of polynomial multiplication, therefore, we are actually working with complex $(2n)$th roots of unity.

The FFT

By using a method known as the ***Fast Fourier Transform (FFT)***, which takes advantage of the special properties of the complex roots of unity, we can compute $\mathrm{DFT}_n(a)$ in time $\Theta(n \lg n)$, as opposed to the $\Theta(n^2)$ time of the straightforward method.

The FFT method employs a divide-and-conquer strategy, using the even-index and odd-index coefficients of $A(x)$ separately to define the two new degree-bound $n/2$ polynomials $A^{[0]}(x)$ and $A^{[1]}(x)$:

$$\begin{aligned} A^{[0]}(x) &= a_0 + a_2 x + a_4 x^2 + \cdots + a_{n-2} x^{n/2-1} , \\ A^{[1]}(x) &= a_1 + a_3 x + a_5 x^2 + \cdots + a_{n-1} x^{n/2-1} . \end{aligned}$$

Note that $A^{[0]}$ contains all the even-index coefficients of A (the binary representation of the index ends in 0) and $A^{[1]}$ contains all the odd-index coefficients (the binary representation of the index ends in 1). It follows that

$$A(x) = A^{[0]}(x^2) + x A^{[1]}(x^2) , \tag{32.9}$$

so that the problem of evaluating $A(x)$ at $\omega_n^0, \omega_n^1, \ldots, \omega_n^{n-1}$ reduces to

1. evaluating the degree-bound $n/2$ polynomials $A^{[0]}(x)$ and $A^{[1]}(x)$ at the points

   $$(\omega_n^0)^2, (\omega_n^1)^2, \ldots, (\omega_n^{n-1})^2 , \tag{32.10}$$

   and then
2. combining the results according to equation (32.9).

By the halving lemma, the list of values (32.10) consists not of n distinct values but only of the $n/2$ complex $(n/2)$th roots of unity, with each root occurring exactly twice. Therefore, the polynomials $A^{[0]}$ and $A^{[1]}$ of degree-bound $n/2$ are recursively evaluated at the $n/2$ complex $(n/2)$th roots of unity. These subproblems have exactly the same form as the original problem, but are half the size. We have now successfully divided an n-element DFT_n computation into two $n/2$-element $\mathrm{DFT}_{n/2}$ computations. This decomposition is the basis for the following recursive FFT algorithm, which computes the DFT of an n-element vector $a = (a_0, a_1, \ldots, a_{n-1})$, where n is a power of 2.

RECURSIVE-FFT(a)

```
1  n ← length[a]              ▷ n is a power of 2.
2  if n = 1
3     then return a
4  ω_n ← e^{2πi/n}
5  ω ← 1
6  a^[0] ← (a_0, a_2, ..., a_{n-2})
7  a^[1] ← (a_1, a_3, ..., a_{n-1})
8  y^[0] ← RECURSIVE-FFT(a^[0])
9  y^[1] ← RECURSIVE-FFT(a^[1])
10 for k ← 0 to n/2 - 1
11      do y_k ← y_k^[0] + ω y_k^[1]
12         y_{k+(n/2)} ← y_k^[0] - ω y_k^[1]
13         ω ← ω ω_n
14 return y                   ▷ y is assumed to be column vector.
```

The RECURSIVE-FFT procedure works as follows. Lines 2–3 represent the basis of the recursion; the DFT of one element is the element itself, since in this case

$$\begin{aligned} y_0 &= a_0\,\omega_1^0 \\ &= a_0 \cdot 1 \\ &= a_0 \,. \end{aligned}$$

Lines 6–7 define the coefficient vectors for the polynomials $A^{[0]}$ and $A^{[1]}$. Lines 4, 5, and 13 guarantee that ω is updated properly so that whenever lines 11–12 are executed, $\omega = \omega_n^k$. (Keeping a running value of ω from iteration to iteration saves time over computing ω_n^k from scratch each time through the **for** loop.) Lines 8–9 perform the recursive $\mathrm{DFT}_{n/2}$ computations, setting, for $k = 0, 1, \ldots, n/2 - 1$,

$$\begin{aligned} y_k^{[0]} &= A^{[0]}(\omega_{n/2}^k) \,, \\ y_k^{[1]} &= A^{[1]}(\omega_{n/2}^k) \,, \end{aligned}$$

or, since $\omega_{n/2}^k = \omega_n^{2k}$ by the cancellation lemma,

$$\begin{aligned} y_k^{[0]} &= A^{[0]}(\omega_n^{2k}) \,, \\ y_k^{[1]} &= A^{[1]}(\omega_n^{2k}) \,. \end{aligned}$$

Lines 11–12 combine the results of the recursive $\mathrm{DFT}_{n/2}$ calculations. For $y_0, y_1, \ldots, y_{n/2-1}$, line 11 yields

$$\begin{aligned} y_k &= y_k^{[0]} + \omega_n^k y_k^{[1]} \\ &= A^{[0]}(\omega_n^{2k}) + \omega_n^k A^{[1]}(\omega_n^{2k}) \\ &= A(\omega_n^k) \,, \end{aligned}$$

where the last line of this argument follows from equation (32.9). For $y_{n/2}, y_{n/2+1}, \ldots, y_{n-1}$, letting $k = 0, 1, \ldots, n/2 - 1$, line 12 yields

$$\begin{aligned}
y_{k+(n/2)} &= y_k^{[0]} - \omega_n^k y_k^{[1]} \\
&= y_k^{[0]} + \omega_n^{k+(n/2)} y_k^{[1]} \\
&= A^{[0]}(\omega_n^{2k}) + \omega_n^{k+(n/2)} A^{[1]}(\omega_n^{2k}) \\
&= A^{[0]}(\omega_n^{2k+n}) + \omega_n^{k+(n/2)} A^{[1]}(\omega_n^{2k+n}) \\
&= A(\omega_n^{k+(n/2)}) .
\end{aligned}$$

The second line follows from the first since $\omega_n^{k+(n/2)} = -\omega_n^k$. The fourth line follows from the third because $\omega_n^n = 1$ implies $\omega_n^{2k} = \omega_n^{2k+n}$. The last line follows from equation (32.9). Thus, the vector y returned by RECURSIVE-FFT is indeed the DFT of the input vector a.

To determine the running time of procedure RECURSIVE-FFT, we note that exclusive of the recursive calls, each invocation takes time $\Theta(n)$, where n is the length of the input vector. The recurrence for the running time is therefore

$$\begin{aligned}
T(n) &= 2T(n/2) + \Theta(n) \\
&= \Theta(n \lg n) .
\end{aligned}$$

Thus, we can evaluate a polynomial of degree-bound n at the complex nth roots of unity in time $\Theta(n \lg n)$ using the Fast Fourier Transform.

Interpolation at the complex roots of unity

We now complete the polynomial multiplication scheme by showing how to interpolate the complex roots of unity by a polynomial, which enables us to convert from point-value form back to coefficient form. We interpolate by writing the DFT as a matrix equation and then looking at the form of the matrix inverse.

From equation (32.4), we can write the DFT as the matrix product $y = V_n a$, where V_n is a Vandermonde matrix containing the appropriate powers of ω_n:

$$\begin{pmatrix} y_0 \\ y_1 \\ y_2 \\ y_3 \\ \vdots \\ y_{n-1} \end{pmatrix} = \begin{pmatrix} 1 & 1 & 1 & 1 & \cdots & 1 \\ 1 & \omega_n & \omega_n^2 & \omega_n^3 & \cdots & \omega_n^{n-1} \\ 1 & \omega_n^2 & \omega_n^4 & \omega_n^6 & \cdots & \omega_n^{2(n-1)} \\ 1 & \omega_n^3 & \omega_n^6 & \omega_n^9 & \cdots & \omega_n^{3(n-1)} \\ \vdots & \vdots & \vdots & \vdots & \ddots & \vdots \\ 1 & \omega_n^{n-1} & \omega_n^{2(n-1)} & \omega_n^{3(n-1)} & \cdots & \omega_n^{(n-1)(n-1)} \end{pmatrix} \begin{pmatrix} a_0 \\ a_1 \\ a_2 \\ a_3 \\ \vdots \\ a_{n-1} \end{pmatrix} .$$

The (k, j) entry of V_n is ω_n^{kj}, for $j, k = 0, 1, \ldots, n-1$, and the exponents of the entries of V_n form a multiplication table.

For the inverse operation, which we write as $a = \mathrm{DFT}_n^{-1}(y)$, we proceed by multiplying y by the matrix V_n^{-1}, the inverse of V_n.

Theorem 32.7
For $j, k = 0, 1, \ldots, n-1$, the (j, k) entry of V_n^{-1} is ω_n^{-kj}/n.

Proof We show that $V_n^{-1} V_n = I_n$, the $n \times n$ identity matrix. Consider the (j, j') entry of $V_n^{-1} V_n$:

$$\begin{aligned}[V_n^{-1} V_n]_{jj'} &= \sum_{k=0}^{n-1} (\omega_n^{-kj}/n)(\omega_n^{kj'}) \\ &= \sum_{k=0}^{n-1} \omega_n^{k(j'-j)}/n .\end{aligned}$$

This summation equals 1 if $j' = j$, and it is 0 otherwise by the summation lemma (Lemma 32.6). Note that we rely on $-(n-1) < j' - j < n-1$, so that $j' - j$ is not divisible by n, in order for the summation lemma to apply. ∎

Given the inverse matrix V_n^{-1}, we have that $\text{DFT}_n^{-1}(y)$ is given by

$$a_j = \frac{1}{n} \sum_{k=0}^{n-1} y_k \omega_n^{-kj} \tag{32.11}$$

for $j = 0, 1, \ldots, n-1$. By comparing equations (32.8) and (32.11), we see that by modifying the FFT algorithm to switch the roles of a and y, replace ω_n by ω_n^{-1}, and divide each element of the result by n, we compute the inverse DFT (see Exercise 32.2-4). Thus, DFT_n^{-1} can be computed in $\Theta(n \lg n)$ time as well.

Thus, by using the FFT and the inverse FFT, we can transform a polynomial of degree-bound n back and forth between its coefficient representation and a point-value representation in time $\Theta(n \lg n)$. In the context of polynomial multiplication, we have shown the following.

Theorem 32.8 (Convolution theorem)
For any two vectors a and b of length n, where n is a power of 2,

$$a \otimes b = \text{DFT}_{2n}^{-1}(\text{DFT}_{2n}(a) \cdot \text{DFT}_{2n}(b)) ,$$

where the vectors a and b are padded with 0's to length $2n$ and $\cdot$ denotes the componentwise product of two $2n$-element vectors. ∎

Exercises

32.2-1
Prove Corollary 32.4.

32.2-2
Compute the DFT of the vector $(0, 1, 2, 3)$.

32.2-3

Do Exercise 32.1-1 by using the $\Theta(n \lg n)$-time scheme.

32.2-4

Write pseudocode to compute DFT_n^{-1} in $\Theta(n \lg n)$ time.

32.2-5

Describe the generalization of the FFT procedure to the case in which n is a power of 3. Give a recurrence for the running time, and solve the recurrence.

32.2-6 ★

Suppose that instead of performing an n-element FFT over the field of complex numbers (where n is even), we use the ring $\mathbf{Z}_m$ of integers modulo m, where $m = 2^{tn/2} + 1$ and t is an arbitrary positive integer. Use $w = 2^t$ instead of ω_n as a principal nth root of unity, modulo m. Prove that the DFT and the inverse DFT are well defined in this system.

32.2-7

Given a list of values $z_0, z_1, \ldots, z_{n-1}$ (possibly with repetitions), show how to find the coefficients of the polynomial $P(x)$ of degree-bound n that has zeros only at $z_0, z_1, \ldots, z_{n-1}$ (possibly with repetitions). Your procedure should run in time $O(n \lg^2 n)$. (*Hint:* The polynomial $P(x)$ has a zero at z_j if and only if $P(x)$ is a multiple of $(x - z_j)$.)

32.2-8 ★

The ***chirp transform*** of a vector $a = (a_0, a_1, \ldots, a_{n-1})$ is the vector $y = (y_0, y_1, \ldots, y_{n-1})$, where $y_k = \sum_{j=0}^{n-1} a_j z^{jk}$ and z is any complex number. The DFT is therefore a special case of the chirp transform, obtained by taking $z = \omega_n$. Prove that the chirp transform can be evaluated in time $O(n \lg n)$ for any complex number z. (*Hint:* Use the equation

$$y_k = z^{k^2/2} \sum_{j=0}^{n-1} \left(a_j z^{j^2/2}\right) \left(z^{-(k-j)^2/2}\right)$$

to view the chirp transform as a convolution.)

32.3 Efficient FFT implementations

Since the practical applications of the DFT, such as signal processing, demand the utmost speed, this section examines two efficient FFT implementations. First, we shall examine an iterative version of the FFT algorithm that runs in $\Theta(n \lg n)$ time but has a lower constant hidden in the Θ-notation than the recursive implementation in Section 32.2. Then, we shall use the insights that led us to the iterative implementation to design an efficient parallel FFT circuit.

Da bismo ilustrirali primjene u obradi signala trebamo još neka svojstva transformacije DFT_n.

Imajući u vidu periodička proširenja vektora a, y, možemo uvesti pojmove parnosti i neparnosti

- Kažemo da je vektor $a \in \mathbb{C}^n$ paran, ako vrijedi

$$a_j = a_{n-j} \quad , \quad j = ((0)), 1, \dots, n-1$$

U periodičkom proširenju s periodom n, zbog $a_j = a_{n+j}$, to odgovara poznatoj relaciji

$$a_j = a_{-j} \quad , \quad \forall j$$

Analogno, $a \in \mathbb{C}^n$ je neparan, ako je

$$a_j = -a_{n-j} \quad , \quad j = 0, 1, \dots, n-1$$

što odgovara

$$a_j = -a_{-j} \, , \quad \forall j .$$

- DFT_n čuva parnost i neparnost, jer vrijedi:

$$a \begin{Bmatrix} \text{paran} \\ \text{neparan} \end{Bmatrix} \Rightarrow y = DFT_n(a) \begin{Bmatrix} \text{paran} \\ \text{neparan} \end{Bmatrix}$$

Naravno, vrijedi i obrat.

- Sasvim općenito, DFT_n je definiran na $\mathbb{C}^n$. Ako znamo da je vektor realan, što vrijedi za njegovu transformaciju?

Za par $y = DFT_n(a)$, $a = DFT_n^{-1}(y)$ vrijedi:

$$a \text{ realan} \Rightarrow y_k = \overline{y}_{n-k} = \overline{y}_{-k} \quad , \quad \forall k$$

$$y \text{ realan} \Rightarrow a_j = \overline{a}_{n-j} = \overline{a}_{-j} \quad , \quad \forall j$$

(potez = konjugiranje)

- Kad ova dva rezultata spojimo zajedno, dobivamo

$$a \begin{Bmatrix} \text{realan i paran} \\ \text{realan i neparan} \\ \text{imaginaran i paran} \\ \text{imaginaran i neparan} \end{Bmatrix} \begin{matrix} \Rightarrow \\ \Leftarrow \end{matrix} \; y \begin{Bmatrix} \text{realan i paran} \\ \text{imaginaran i neparan} \\ \text{imaginaran i paran} \\ \text{realan i neparan} \end{Bmatrix}$$

Drugim riječima:

paranost čuva realan, imaginaran
neparnost mijenja realan $\leftrightarrow$ imaginaran.

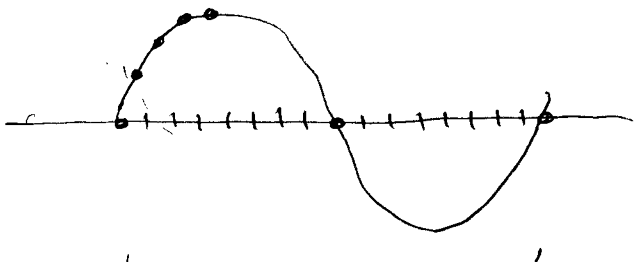
- Dakle, ako želimo raditi s realnim nizom a i ostati u realnoj domeni za $y = DFT_n(a)$ onda treba a proširiti po parnosti do a', napraviti $DFT_{2n}(a') = y'$ i ignorirati drugu polovinu od y'.

  Ovaj "trik" se vrlo često koristi u obradi signala. Naravno, druga parna polovina se ne mora računati.

- "Pristojne" signale obično zamišljamo kao kombinacije valova - cosinusa i sinusa s nekim amplitudama (faktor uz funkciju) i nekim frekvencijama (argument funkcije).

Od takve linearne kombinacije - koja ima oblik trigonometrijskog polinoma - tj. funkcije definirane na vremenskoj kontinuiranoj domeni mi uzimamo <u>uzorak</u> u diskretnim vremenskim točkama.

Periodičnost postižemo tako da je frekvencija uzimanja uzoraka = višekratnik perioda signala tj. višekratnik najviše frekvencije u signalu.



cjelobrojni broj podintervala

- Vrijednosti signala (funkcije) u točkama uzimanja uzorka su vrijednosti u vektoru a.

- skip bit reversal (aho ide par!)
- efficient bit-reversal ?

- Efficient iterative FFT + butterfly + parallel:
  CLR, pp. 791-796, Demmel, Lect. 14, pp. 4-8
  - Give operation counts M, A in $\mathbb{C}$, in $\mathbb{R}$ (Wilf, p. 88)
    (without, and with table lookup)

- General FFT for any $n \in \mathbb{N}$
  Henrici 3, pp. (2-6), 7-9
  Wilf, pp. (88), 89-94
  + improved alg. via convolution: (convolution time / general FFT_n time)
  Henrici 3, §13.7, pp. 54-59

- Time series, smoothing + filtering:
  intro: Henrici 3, pp. 59-61
  example: Demmel, Lect 13, pp. 3-5
  Lect 15, pp. 2-4
  Matlab fftdemo.

- Complex applications of FFT:
  - review polynomial mult. + oper. count
    Henrici 3, pp. 64-65
  - Fast algorithms for FPS, posebno polynomial division + remainder
    Henrici 3, pp. 71-79 [za uvod u FPS → Henrici 1]
  - Chirp transform in $O(n \log n)$
    CLR, Ex. 32.2-8, p. 791
    Henrici 3, Prob. (13.9.) 1, p. 79
  - General polynomial evaluation and interpolation
    at $z_1, -, z_n$ (or $z_0, -, z_{n-1}$)
    CLR, Ex. 32.2-7, p. 791
    Horowitz-Sahni, ALG,

Applications of FFT:

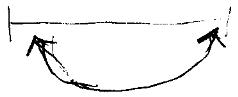
General eval/interpol. in $O(n(\log n)^2)$ [Horowitz–Sahni ALG]
Spec. cases – chirp (Eval – CLR, 32-4, pp. 798–799)

– L.., Dahl – all derivatives, Shaw-Traub (CLR, 32-3, p. 798)
– Schönhage–Strassen

– Modular arithmetic – $\downarrow \rightarrow \uparrow$ (HS-ALG)
FFT in this context (CLR, 32-5, p. 799)

Oro u number-theoretic alg's + Chinese rem. th.
+ Extended Euclid

– Kada je stvar realna i kako to postići (simetrija!)
(Weaver?)
p. 248, 254–256



$$f = DFT_n(F) = \mathcal{F}^{-1}(F)$$
$$F = DFT_n^{-1}(f) = \mathcal{F}(f)$$
tj. $F(j) = \mathcal{F}(f(k)),\ f(k) = \mathcal{F}^{-1}(F(j))$

– $y = DFT_n(a)$ → a_j su koef. u Four. redu funkcije
↳ evaluation at $\omega_n^k,\ k=0,\dots,n-1$
(obtain a time-domain sample)
restauracija a_j → interpolation (DFT_n^{-1})

Iz ovog promatranja rada rekurzivnog FFT algoritma dobivamo sljedeće zaključke za konstrukciju iterativnog FFT algoritma:

1. Iz "djeca prije čvora roditelja" $\Rightarrow$
   [nije nužno, ali je zgodno napraviti takvu organizaciju posla]:
   stablo obrađujemo po slojevima – nivoima iste dubine/visine i to od <u>dna</u> prema <u>vrhu</u> ↑.
   Dakle, prvo obradimo <u>sve</u> listove (su DFT_1), pa onda sve čvorove iznad njih (su DFT_2) i tako redom, do konjena (traženi DFT_n, $n = 2^m$).
   Tj. ovo je vanjska petlja, koja prolazi slojeve odozdo prema gore:

   <u>for</u> $s := 1$ <u>to</u> m <u>do</u> (s = stage = stadij = sloj)
   obradi sloj s;

2. Čvorovi na istom sloju <u>ne ovise</u> jedan o drugom
   $\Rightarrow$ potpuno je <u>svejedno</u> kojim redom obrađujemo čvorove na istom sloju.
   Tih čvorova na "visini" s ima $n/2^s$

<u>1 polje y = radno</u> (izlaz $y_k = y[k]$
$\Rightarrow P_n = \text{bit-rev}_n$)
$P_n = P_n^{-1}$!

– Inner 2 loops
$\Leftrightarrow$ $n/2$ butterfly op's!

Paral. time = $O(\log n) \times n$ "leptir proc."

Seq. time/op's
$\frac{n}{2} \log n \; M_{\mathbb{C}} = 4 M_R \times 2 A_R$
$n \log n \; A_{\mathbb{C}} = 2 A_R$
$\Bigg\}$ $2n \log n \; M_{\mathbb{R}}$, $3n \log n \; A_{\mathbb{R}}$

$\Sigma = \frac{3}{2} n \lg n$ nad $\mathbb{C}$

$\Sigma = 5n \lg n$ nad $\mathbb{R}$

Uočiti: – fix bitova (od vrha prema dnu – po nivoima stabla) ide straga – od kraja ← prema naprijed

– što znači blok x-ova sprijeda (ispred fiksiranih bitova): tu treba redom pometati sve binarne zapise (→ rastuće po brojevima) brojeva od $\emptyset$ do $2^{(\text{broj x-ova})}-1$, – tj. ~~x bitove~~
$\emptyset \ldots \emptyset\emptyset, \emptyset \ldots \emptyset 1, 0 \ldots 1\emptyset, \ldots, 11 \ldots 1$.
(to je tako u svakom čvoru!)

– Za iterativni alg:

(X) — mora prvo SVI listovi, pa SVI na nivou iznad i tako redom po nivoima (SVI na istom) odozdo ↑ prema gore.
= 2 petlje: vanjska = nivoi odozdo ↑ gore
unutarnja = obiđi sve čvorove na danom nivou.

– Za vanjsku – uvedi neki brojač ($\emptyset \to m-1$ ili $1 \ldots m$)
– Za unutarnju – zapravo je APSOLUTNO SVEJEDNO kojim redom idem po tim čvorovima (na istom nivou)
– moram malo paziti koji drugi ispod kombiniram u koga – ali to se vidi iz bit-paterna (razlika $\emptyset \leftrightarrow 1$ na bitu koji odgovara broju nivoa odozdo ↑ a bitove brojim sprijeda – kad se penjem)

– Prva varijanta – ide kao što piše – slijeva udesno samo još prvo preuredi a-ove u taj poredak. (CLR + loop transpose)

– Sad o poretku (~~kako~~ bit-reversal, kako ga generirati, rekurz. alg., najbolje je pre-compute i store, kao za potencije od ω_n).

– Par $k, k+\frac{n}{2}$ (n na nivou) u odd-even (----0---- / ----1----) [DEMMEL]
– Na kraju – (B) izbaci reversal (par DFT, DFT^{-1} ~~sprijeda~~ straga, sprijeda = cancel!)
(A) okreni stvar – bit-reversal na kraju (v. DFT^{-1} ili half-half splitting)

$$A = A_1 + x^{n/2} \cdot A_2$$

(X) JEDINI problem na samom DNU – DFT_1 = kopiraj – ali odakle kamo – zato je zgodno možda napraviti PRE-COPY $a \mapsto y$
– Tek IZA ide ~~ko~~ COMBINE 2x DFTpola u DFTcijeli.

(*) Kad već pamtiš ω_n^k, zapamti ih u bit-reverse !? (X)

Ova komplementarnost znači samo to da je zbroj centralno simetričnog para jednak $n-1 = 2^m - 1$ (broj koji ima sve bitove jednake 1).

Međutim, to nam neće puno pomoći da dobijemo kako izgleda poredak bitova argumenta od FFT_1 na danom mjestu na dnu stabla.

- C.C. – u jednom polju (da se izbjegne kopiranje na stacku)

- inner 2 loops ⟺ n/2 butterfly op's.

Brza iterativna varijanta algoritma za DFT_n, $n=2^m$

U daljnjem stalno pretpostavljamo da je n potencija od 2, tj. $n=2^m$, $m\in\mathbb{N}$.

Da bismo dobili iterativnu varijantu algoritma iz rekurzivne, treba "raspakirati" rekurziju, tj. prvo pogledati što se događa na pojedinim nivoima rekurzije.

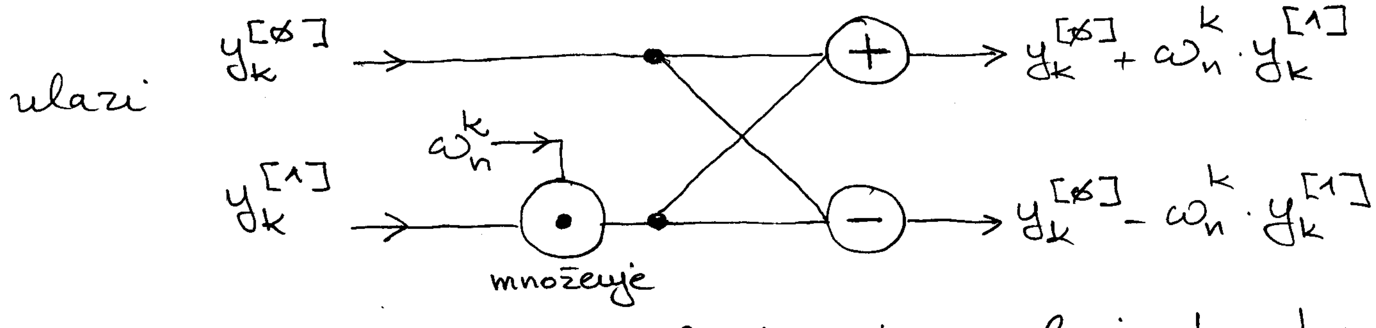
Rekurzivni algoritam ima tipični oblik obilaska binarnog stabla i to u "post"- uređaju – prvo obilazimo oba djeteta, obavljamo neki posao i vraćamo se natrag.

Naš "neki" posao je tzv. "butterfly" ili leptir operacija u unutarnjoj petlji, nakon rekurzivnih poziva.

Tu operaciju možemo shematski prikazati kao kombinatorni ili elektronički sklop (engl. "circuit) koji iz 2 ulaza $y_k^{[0]}$ i $y_k^{[1]}$, uz zadani faktor ω_n^k, generira 2 izlaza zadana (definirana) relacijama

$$y_k = y_k^{[0]} + \omega_n^k \cdot y_k^{[1]}$$
$$y_{k+n/2} = y_k^{[0]} - \omega_n^k \cdot y_k^{[1]}.$$

Standardna shema je:



Ove krugove možemo zamisliti i kao vrlo jednostavne procesore (krugove), koje možemo međusobno vezati da napravimo računalo ("veliki" krug) za računanje DFT_n.

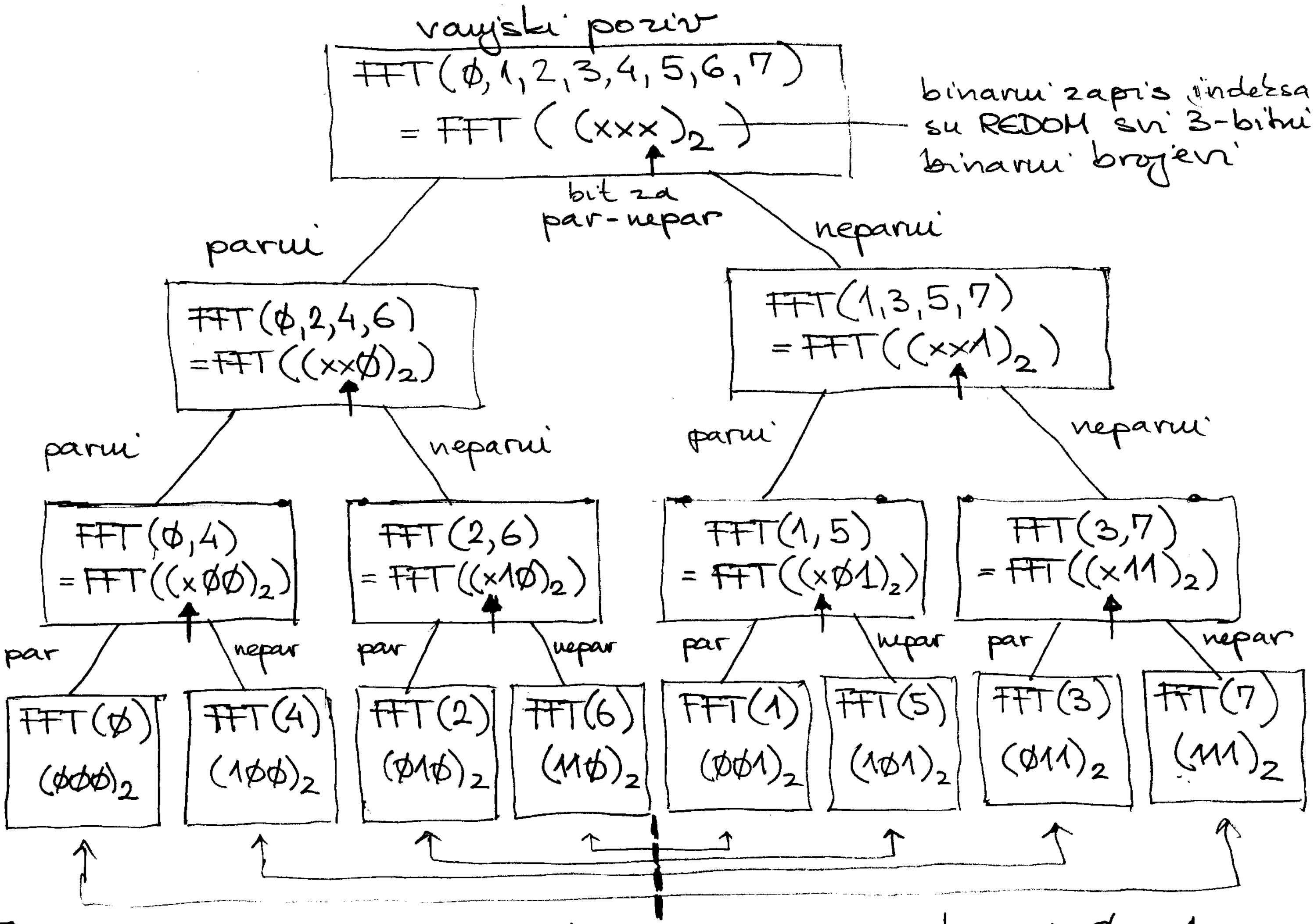
Pogledajmo sad kako izgleda stablo koje generiramo i obilazimo u rekurzivnom FFT algoritmu za DFT_n.

Ključna operacija je rastav ulaznog vektora a na podvektore $a^{[0]}$, $a^{[1]}$ koji sadrže komponente parnih, odnosno, neparnih indeksa iz a.

Parnost indeksa je određena zadnjim bitom u njegovom binarnom prikazu, pa je zgodno indekse prikazivati i kao m-bitne brojeve u bazi 2.

Nacrtajmo stablo poziva za vanjski poziv s $n=8=2^3$ tj. za računanje $DFT_8((a_0,a_1,a_2,a_3,a_4,a_5,a_6,a_7)^T)$

U čvorovima stabla pišemo, radi jednostavnosti, samo indekse polaznih komponenti vektora a, koje dolaze na ulaz u FFT rekurziji.



NE OVDJE

Uočimo binarne zapise na dnu – oni su simetrični $\emptyset \Leftrightarrow 1$ na istom mjestu, oko centralne osi!

Na istoj udaljenosti lijevo i desno od te osi, binarni zapisi su komplementarni – imaju sve bitove obratne (mjesto, po mjesto $0 \Leftrightarrow 1$).

Ovo stablo prikazuje samo rekurzivni dio DFT_n algoritma. Cijeli rekurzivni algoritam radi ovako

- obiđi lijevo i desno (parno i neparno) podstablo
- zatim, u čvoru izračunaj kombinaciju ta dva podstabla ($2 \times DFT_{pola} \to DFT_{cijeli}$)

Dakle, u svakom čvoru je još "sakrivena" ona petlja s dna rekurzivnog algoritma. Drugim riječima, rekurzivni DFT_n odgovara obilasku ovog stabla u post poretku (lijevo dijete, desno dijete, čvor).

- Uočimo da je zapravo svejedno da li prvo obilazimo lijevo ili desno podstablo, bitno je samo da je čvor na kraju. Tj. prvo djeca, pa čvor (roditelj te djece).

- Objasnimo još što znače oznake u binarnom zapisu indeksa koji ulaze u FFT.

$(xxx)_2$ u korijenu znači da su indeksi parametara od FFT svi 3-bitni binarni brojevi (svaki x može biti ∅ ili 1) - dakle svih 8 takvih brojeva i to rastuće poredani (kao brojevi) ili leksikografski rastuće (kao 3-bitni binarni brojevi. Dakle, 0, 1, 2, ..., 7.

- svaki ulazak u parno ili neparno podstablo fiksira prvi nefiksirani tj. slobodni bit x na ∅ ili 1, i to sa stražnje strane prikaza (od znamenke jedinica prema naprijed: ⟵).
Dakle, lijevo-parno dijete korijena ima indekse čiji 3-bitni prikaz je $(xx\emptyset)_2$ - svi parni!
Analogno, desno-neparno dijete ima indekse čiji 3-bitni prikaz je $(xx1)_2$ - svi neparni.

- Dakle, pri spuštanju od vrha prema dnu, tj. od korijena prema listovima, fiksiranje bitova ide sa stražnje strane prema naprijed:

prvi slobodni x ↗ ∅ u lijevo-parno dijete
↘ 1 u desno-neparno dijete

- Što znači blok x-ova sprijeda, ispred fiksnih (fiksiranih) bitova?

Tu treba redom – leksikografski rastuće, pometati sve binarne zapise u kojima se svaki pojedini x može zamijeniti s Ø ili 1, tj. supstituirati redom ØØ...Ø, ØØ...1, ..., 11...1

(binarni prikazi brojeva od Ø do $2^{(\text{broj x-ova})}-1$)

- Naravno, to vrijedi u svakom čvoru:
broj x-ova ⟺ broj parametara je $2^{\text{broj x-ova}}$.

- Na samom dnu, svi bitovi su fiksirani ⟺ zovemo FFT s jednim jedinim parametrom, ili, preciznije, zovemo FFT na vektoru duljine 1 – na jednom elementu polaznog vektora.

Taj dio posla je trivijalan – svodi se na golo kopiranje vrijednosti. Jedino je pitanje odakle-kamo? Otkud to?!

- U naše stablo smo ugradili sve elemente rekurzivnog algoritma osim jednog!

Što imamo:
2 rekurzivna poziva → 2 djeteta
kombinacija 2×DFT_{pola} ↦ DFT_{cijeli} → petlja u čvoru.

Što fali?

Ono kopiranje nizova prije rekurzivnih poziva, koje smo elegantno izbjegli u stablu, jer eksplicitno pišemo parametre!

No, općenito, takav ulazni parametar je zapravo <u>lokalna</u> kopija na <u>stacku</u>, kod izvršavanja rekurzije!

Vrlo zgodno bi bilo ako možemo izbjeći to gomilanje nizova na stacku i sve raditi na <u>jednom globalnom vektoru</u>.

Naime, ako želimo iterativni algoritam, onda
- ili moramo simulirati stack za rekurziju
- ili moramo raditi na jednom objektu (taj bi bio globalan u rekurziji!)

Iz ovog promatranja rada rekurzivnog FFT algoritma dobivamo sljedeće zaključke za konstrukciju iterativnog FFT algoritma.

1. Iz "djeca prije čvora-roditelja" $\Rightarrow$ (stvarno je $\Leftarrow$)
   mogli bismo to realizirati tako da bude
   "<u>sva</u> djeca prije (svih) čvorova-roditelja"
   [Sad je jasno zašto stvarno vrijedi $\Leftarrow$, a ne $\Rightarrow$].
   Naravno, to nije nužno, ali je zgodno napraviti baš takvu organizaciju posla.

- Stablo obrađujemo po slojevima-nivoima iste dubine/visine, i to od <u>dna</u> prema <u>vrhu</u> ($\uparrow$).

Drugim riječima, prvo obradimo sve listove – tj. nađemo sve DFT_1, pa onda sve čvorove iznad njih (svi DFT_2), i tako redom, do korijena, gdje na kraju nađemo traženi DFT_n, $n = 2^m$.

- Ovime dobivamo <u>vanjsku</u> petlju iterativnog algoritma koja prolazi sve slojeve odozdo prema gore:

```
for s := ∅ to m do
    obradi sloj s;
```

Varijabla s = stage = stadij = sloj.
Brojanje od $\emptyset$ do m odgovara visini od dna (listovi su na visini $\emptyset$). Možemo odmah uočiti da s odgovara i broju x-ova, tj. broju slobodnih bitova na tom sloju.

Svi čvorovi na sloju s imaju s slobodnih bitova, tj. pripadni FFT na ulazu ima niz duljine

$$2^s, \quad s = \emptyset, \ldots, m.$$

- Osim toga, listovi ($s = \emptyset$) su posebno jednostavni. Pripadni DFT_1 u čvoru <u>nema</u> aritmetičkih operacija.
Ako sav posao želimo obaviti u jednom polju y, koje će, na kraju, biti i izlazni rezultat, onda na ovom mjestu – u listovima prebacujemo a-ove u y-e. Kasnije, sve radimo na polju y.

Sve operacije u listovima su oblika

$$y_{neki\_y} := DFT_1(a_{neki\_a}) = a_{neki\_a}.$$

Uočimo odmah da element od a ne moramo prebaciti na isto mjesto u vektor y, tj. može biti

$$neki\_y \neq neki\_a.$$

Bitno je da listova ima točno $n = 2^m$ i da svaki ima svoj element od a. I, očito, dva različita elementa od a ne smiju na isto mjesto u vektor y (jednog od njih - ranijeg - bismo izgubili).

Dakle, obrada svih listova se svodi na operaciju kopiranja vektora a u vektor y, ali ne nužno u istom poretku indeksa, već ih smijemo i permutirati, ako nam to više odgovara za ostatak algoritma!

Neka je p_n ta izabrana permutacija indeksa $0, \ldots, n-1$.

$$j \longmapsto p_n(j), \quad j = 0, \ldots, n-1.$$

Operacija "obradi sloj 0 (listove) ima onda oblik

```
for j := 0 to n-1 do
    y[pn(j)] := a_j;
```

ili $y[j] := a_{p_n^{-1}(j)}$

Ovdje je y radno polje, a ne izlazni vektor, pa indeksiranje pišemo Pascalski - u [ ], da ne dođe do zabune.

Kako treba izabrati permutaciju p_n?
O tome malo kasnije, kad razradimo ostatak iterativnog algoritma.

Ostatak algoritma (nakon ovog kopiranja s permutiranjem) kojeg moramo još razraditi je obrada svih ostalih slojeva:

```
for s := 1 to m do
    obradi sloj s;
```

2. Čvorovi na istom sloju, očito, ne ovise jedan o drugom $\Rightarrow$ potpuno je svejedno kojim redom obrađujemo čvorove na istom sloju.

Tih čvorova na "visini" s ima točno $n/2^s$, ili 2^{m-s}. Dakle, sljedeća petlja bi mogla imati ovih 2^{m-s} prolaza za obradu čvor-po-čvor, u nekom zgodnom poretku. (Nećemo još napisati tu petlju!)

Na kraju, obrada u svakom čvoru je kombinacija

$$2 \times DFT_{2^{s-1}}(\dots) \longmapsto DFT_{2^s}(\dots)$$

a to se svodi na točno 2^{s-1} "leptir" operacija (treća - zadnja petlja).

Odmah možemo uočiti da ove dvije petlje zajedno imaju

$$2^{m-s} \cdot 2^{s-1} = 2^{m-1} = \frac{n}{2}$$

leptir operacija. Te leptir operacije iz starog stanja polja y produciraju novo stanje tog istog radnog polja y

stari y sadrži 2^{m-s+1} podvektora oblika $DFT_{2^{s-1}}$ tj. 2^{m-s+1} "vektorčića" koji su već izračunati DFT-ovi svih podnizova duljine 2^{s-1}.

novi y sadrži samo 2^{m-s} podvektora koji su DFT-ovi svih podnizova duljine 2^s

Naravno, podnizovi su ovi DFT-ovi iz stabla

$$FFT((\underbrace{\underbrace{x\dots x}_{s-1}\overset{fix}{\sqcup\sqcup\dots\sqcup}}_{\text{duljina } 2^{s-1}})_2) \longrightarrow FFT((\underbrace{\underbrace{x\dots xx}_{s}\overset{fix}{\sqcup\dots\sqcup}}_{\text{duljina } 2^{s}})_2).$$

i to

$$\left.\begin{array}{l}(x\dots x\,0\,\sqcup\dots\sqcup)_2 \\ (x\dots x\,1\,\sqcup\dots\sqcup)_2\end{array}\right\} \to (x\dots xx\sqcup\dots\sqcup)_2$$

Naravno, jedino je pitanje gdje unutar polja y se nalaze potrebna 2 vektora koje treba kombinirati i gdje treba smjestiti njihovu kombinaciju.

- Prirodno je da kombinaciju prepišemo preko istih onih mjesta gdje su ranije bili kraći "ulazni" nizovi za kombinaciju.

Čak malo jače od toga!

Osnovna operacija unutar svih petlji je jedna leptir operacija. Nju grubo možemo interpretirati u obliku

2 stara y-a ⟼ 2 nova y-a
(2 elementa) (2 elementa)

Naravno, bilo bi zgodno nova 2 elementa prepisati preko stara 2 i tako to $n/2$ puta!

Na taj način osiguravamo da sve operacije možemo obaviti u istom polju y, bez dodatnih polja i to neovisno o realizaciji obrade pojedinog sloja — tj. 2 unutarnje petlje - po čvorovima i leptirima u čvoru mogu u bilo kom poretku (!) ili čak kao jedna petlja za $n/2$ leptira (!).

Dakle, moramo odlučiti gdje će ti elementi biti u polju y. Naravno, ta odluka ima veze i s tim kako je finalni vektor $y = DFT_n(a)$ poredan u polju y — na izlazu.

Prirodno je izabrati da na izlazu y ima očekivani uređeni poredak

$$y_k = y[k], \quad k = 0, \ldots, n-1.$$

U protivnom, još na kraju moramo permutirati y da ga dobijemo u prirodnom poretku.

[Baš na ovo ćemo se još vratiti!]

Idemo onda, u skladu s tim, organizirati polje y tako da:

(a) svi DFT(...) nizovi imaju prirodan poredak indeksa i to u bloku susjednih lokacija u polju y

(b) prema stablu, DFT-ovi iz lijevih podstabala dolaze ispred onih iz desnih podstabala.

Dakle, parni i neparni koje treba kombinirati dolaze prirodno jedan iza drugog – u bloku

$$DFT_{2^{s-1}}((\underbrace{x..x}_{s-1}\,0\,u...u)_2) \quad DFT_{2^{s-1}}((x...x\,1\,u...u)_2)$$

indeksi elem. u $DFT_{2^{s-1}}$ → $0 \; 1 \; \ldots \; 2^{s-1}-1 \;\; 0 \; 1 \; \ldots \; 2^{s-1}-1$

susjednih 2^s lokacija u y

To znači da polje y na svakom sloju s sadrži točno redom vektore iz čvorova na tom sloju u binarnom stablu i to <u>slijeva udesno</u> (→).

U tom slučaju, vanjske dvije petlje iterativnog algoritma imaju oblik

```
for s := 1 to m do
  for l := 0 to 2^(m-s) - 1 do
    [ p := l·2^s ;  { početni indeks u y }
      kombiniraj 2 DFT-a duljine 2^(s-1) koji
      se nalaze u
          y[p .. p+2^(s-1)-1]  i
          y[p+2^(s-1) .. p+2^s-1]
      u jedan DFT duljine 2^s u
          y[p .. p+2^s-1]
    ]
```

Ova kombinacija ima 2^{s-1} leptira. Svi ti leptiri koriste potencije od ω_{2^s}, jer se računa $DFT_{2^s}(\ldots)$.

Kad uvrstimo pripadne leptir operacije dobivamo kompletni algoritam za penjanje - od prvog sloja $s=1$ do korijena.

Fali nam još samo permutacija p_n za cijeli algoritam.

Da bi "penjanje" radilo, na dnu - u listovima mora vrijediti isto pravilo o poretku DFT_1 u polju y. To znači da u polju y, na početku moraju biti elementi iz vektora a onim redom kojim se pojavljuju u listovima stabla - slijeva udesno.

Za $n=8$, taj poredak je

$$a_0 \; a_4 \; a_2 \; a_6 \; a_1 \; a_5 \; a_3 \; a_7 .$$

Ako pogledamo binarne zapise indeksa u y i indeksa u a, dobivamo tablicu

indeks u y	indeks u a
$0 = 000_2$	$000_2 = 0$
$1 = 001_2$	$100_2 = 4$
$2 = 010_2$	$010_2 = 2$
$3 = 011_2$	$110_2 = 6$
$4 = 100_2$	$001_2 = 1$
$5 = 101_2$	$101_2 = 5$
$6 = 110_2$	$011_2 = 2$
$7 = 111_2$	$111_2 = 7$

Možemo zaključiti da se binarni zapis indeksa u a dobiva <u>obratnim</u> poretkom bitova u binarnom zapisu indeksa u y. Naravno, vrijedi i obratno - dva puta obrnemo redoslijed - ništa se ne mijenja.

Dakle, permutacija p_n radi ovako, za $n=2^m$

ako je binarni zapis indeksa j u a (ili u y)

$$j = (j_{m-1} j_{m-2} \cdots j_1 j_0)_2$$

(j_r su bitovi, $r=0,\ldots,m-1$), onda je binarni zapis indeksa $p_n(j)$ u y (ili u a) oblika

$$p_n(j) = (j_0 j_1 \cdots j_{m-2} j_{m-1})_2 .$$

Ova permutacija p_n koja okreće bitove u obratni poredak zove se bit_reverse (okreni_bitove) permutacija i označava s

$$\text{bit\_rev}_n. \qquad (n = 2^m)$$

Pripadni poredak $\text{bit\_rev}_n(j)$, $j = 0, \ldots, n-1$ zove se obratni bit poredak (bit_reverse order).

Odmah vidimo da je ona sama sebi inverz

$$(\text{bit\_rev}_n)^{-1} = \text{bit\_rev}_n .$$

- Precizan dokaz da je poredak u listovima upravo obratni bit poredak ide ovako.

Pogledajmo korijen stabla. Svi parni indeksi od a idu lijevo – ispred svih neparnih indeksa od a, koji idu desno – straga.

Dakle, ako je zadnji bit j_0 indeksa j za a jednak $\emptyset$ (j paran), onda pripadni a_j ide u prvu polovinu polja y, a svi indeksi u prvoj polovini imaju vodeći bit jednak $\emptyset$

$$j = (j_{m-1} \cdots j_1 \, {}^{0}_{1})_2 \Longleftrightarrow p_n(j) = ({}^{\emptyset}_{1} \, k_{m-2} \cdots k_0)_2$$

Isti princip se rekurzivno ponavlja u svakom čvoru koji nije list, pa indukcijom lako izlazi tvrdnja.
[Precizni zapis koraka indukcije je malo tehnički kompliciran, ali je očito

j_s par/nepar $j_s = {}^{0}_{1} \Longleftrightarrow p_n(j)$ s $j_s = 0$ ispred (<) $p_n(j)$ s $j_s = 1$]

- Da zaključimo. Uz dogovorenu organizaciju polja y prvom sloju ($s = \emptyset$), tj. listovima odgovara operacija

Bit_Reverse_Copy (a) ili (a, y) ili (n, a, y) koja radi sljedeće:

```
function Bit_Reverse_Copy (a);
   n := length(a);
   for j := 0 to n-1 do
      y[ bit_rev_n(j) ] := a_j;
   { može i obratno:  y[j] := a_bit_rev_n(j) }
   endfor;
   return y;
```

Kako ćemo točno realizirati bit_rev_n, o tome malo kasnije (možemo računati za svaki j, ili spremiti unaprijed u vektor).

Prva varijanta iterativnog FFT algoritma je:

```
function FFT_Base (a);  { kompleksni vektor }
   y := Bit_Reverse_Copy(a);   { listovi  s=0 }
   n := length(a);  m := lg(n);   { n=2^m }
   for s := 1 to m do  { slojevi }
      g := 2^s;
      ω_g := e^(2πi/g);
      for l := 0 to n/g - 1 do  { ide do 2^(m-s) - 1 }
         p := l·g;  { start bloka }
         { ω := 1;      - ako računamo sve potencije }
         for k := 0 to g/2 - 1 do  { ide do 2^(s-1) - 1 }
            { leptir operacija }
            t := ω_g^k · y[p + k + g/2];
               { ili  t := ω * y[p+k+g/2]; }
            u := y[p+k];
            y[p+k] := u+t;
            y[p+k+g/2] := u-t;
         { ω := ω * ω_g }
         endfor; {k}
      endfor; {l}
   endfor; {s}
   return y;
```

Ovdje se lijepo vidi da 2^{m-s} puta računamo iste potencije od $\omega_g = \omega_{2^s}$ (ω_g^k, za $k = 0, \dots, 2^{s-1} - 1$). [Naravno, ako zaista računamo te potencije, a ne čitamo iz tablice].

Međutim, to nije neki problem, jer unutarnje dvije petlje možemo bez problema okrenuti – "transponirati" s idejom da za fiksni k obavimo sve leptire koji trebaju ω_g^k.

Ta poboljšana varijanta algoritma je:

```
function FFT_Iter (a);   { kompleksni vektor }
  y := Bit_Reverse_Copy (a);   { listovi s=∅ }
  n := length(a);  m := lg(n);   { n = 2^m }
  for s := 1 to m do   { slojevi }
    g := 2^s;
    ω_g := e^(2πi/g);
    { ω := 1;  - ako računamo potencije. }
    for k := ∅ to g/2-1 do   { ide do 2^(s-1) - 1 }
      for l := ∅ to n/g-1 do   { ide do 2^(m-s) - 1 }
        p := l·g + k;   { umjesto leptira! }
        t := ω_g^k · y[p+g/2];   { ili t := ω * y[p+g/2]; }
          { Okrenem redoslijed operacija i eliminiram u! }
        y[p+g/2] := y[p] - t;
        y[p] := y[p] + t;
      endfor; { l }
    { ω := ω * ω_g;  - ako računamo potencije. }
    endfor; { k }
  endfor; { s }
  return y;
```

Daljnje uštede možemo napraviti samo efikasnijim indeksiranjem. Na primjer, zapamtim $g/2 = 2^{s-1}$ u vanjskoj petlji.

Ako mogu pisati korak u petlji (tj. ne mora biti 1 ili -1), onda još možemo skratiti zapis.

Blok u okolini unutarnje dvije petlje bi tada izgledao ovako:

$g2 := 2^{s-1};$ $\{g/2\}$
for $k := 0$ **to** $g2-1$ **do**
  for $p := k$ **to** $n-1$ **step** g **do**
   ⋮

tako da p prolazi točno vrijednostima

$$k, g+k, 2g+k, \ldots, (n/g-1)\cdot g+k.$$

Naime, zadnja vrijednost je $n-g+k$, pa ako dodam još jednom g, dobijem $n+k \geq \{k\geq 0\} \geq n > n-1$, tj. strogo više od zadnje dozvoljene vrijednosti $n-1$ za p, a to znači da se petlja prekinula prije toga!

Ponovimo: unutarnje dvije petlje zajedno rade točno $n/2$ leptir operacija. Te operacije su potpuno nezavisne, ako uzmemo da se

$$\omega_{2^s}^{k}$$

čita iz tablice. Tu odmah dobivamo i paralelnu implementaciju za DFT_{2^m} [Slika].

– Možemo još "napasti" dva problema:

(1) – Kako se zgodno čitaju potencije $\omega_{2^s}^{k}$ iz unaprijed pripremljene tablice?

Pretpostavimo da smo unaprijed izračunali vektor ili polje omega, duljine n, tako da je

$$\text{omega}[j] = \omega_n^{j}, \quad j = 0, \ldots, n-1$$

(tj. spremili lijepo sve n-te korijene iz jedinice).

Kako ga treba iskoristiti da izbjegnemo bilo kakvo računanje s $\omega_g^k = \omega_{2^s}^k$?

[Napomena: složenost računanja polja omega je linearna u n, preciznije $= n$ kompleksnih množenja ($= 4n$ real. množenja $+ 2n$ real. zbrajanja)

Sjetimo se tzv. formule kraćenja

$$\omega_q^k = \omega_{q\cdot d}^{k\cdot d}, \quad \forall d \in \mathbb{N}$$

i uvažimo $q = 2^s$. Želimo $q \cdot d = 2^m = n$, pa treba uzeti

$$d = \frac{n}{q} = 2^{m-s}.$$

Dakle:

$$\omega_q^k = \omega_n^{k\cdot 2^{m-s}} = \text{omega}\,[k \cdot 2^{m-s}],$$

tj. u naredbi koja računa t iskoristimo omega $[k \cdot 2^{m-s}]$, odnosno omega $[k \cdot n/q]$, jer n/q ionako koristimo u gornjoj granici unutarnje petlje.

Ovo je bilo lako! I još je cijena linearna u n. (uz dodatno polje duljine n)

(2) – Kako se računa bit_rev_n, odnosno kako realizirati operaciju Bit_Reverse_Copy?

To više nije tako jednostavno!

Naravno, ako znamo n unaprijed (i) pretpostavimo da smo unaprijed izračunali vektor Bit_Rev, duljine n

$$\text{Bit\_Rev}[j] = \text{bit\_rev}_n(j), \quad j = 0, \dots, n-1,$$

onda je Bit_Reverse_Copy trivijalan. Samo čitamo potrebne indekse iz polja Bit_Rev i gotovo. Tada čitav Bit_Reverse_Copy ide u vremenu $O(n)$ – tj. opet linearno u n.

Čak ne bismo morali kopirati u novo polje y. Mogli bismo napraviti i operaciju

Bit_Reverse_Perm (a)

koja vrši odgovarajuću permutaciju unutar istog polja a, bez dodatnog vektora – sa samo jednom pomoćnom varijablom. I opet je trošak linearan u n.

Sasvim je drugačije ako Bit_Reverse_$\left\{\begin{matrix}\text{Copy}\\ \text{Perm}\end{matrix}\right\}$ (a) treba realizirati **bez** unaprijed izračunatog polja Bit_Rev.

Što onda? Uostalom, a kako bi se izračunao vektor Bit_Rev?

Odgovor bitno ovisi o tome koje operacije imamo na raspolaganju!

Ako koristimo samo cjelobrojne aritmetičke operacije, onda nam nema spasa. Stvar traje $O(n \log n)$ vremena ili operacija (sekvencijalno).

Prvi očiti način je tako da svaki j dijeljenjem s 2 rastavljamo u bitove, koje onda množenjem u obratnom redu (recimo Hornerom) akumuliramo u $\text{bit-rev}_n(j)$. To traje $c \cdot \lg n$ operacija, jer j ima $\lg n$ bitova. I tako to za svaki j, što je

$$c' n \cdot \lg n \text{ operacija}$$

(ima ponešto mogućnosti za malo ušteda, ali ne bitno).

Drugi način je rekurzivan, a nalikuje na naše FFT-stablo. Nažalost, složenost je istog reda veličine kao i cijeli FFT!

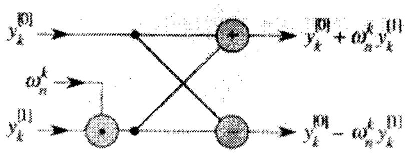


Figure 32.3 A butterfly operation. The two input values enter from the left, ω_n^k is multiplied by $y_k^{[1]}$, and the sum and difference are output on the right. The figure can be interpreted as a combinational circuit.

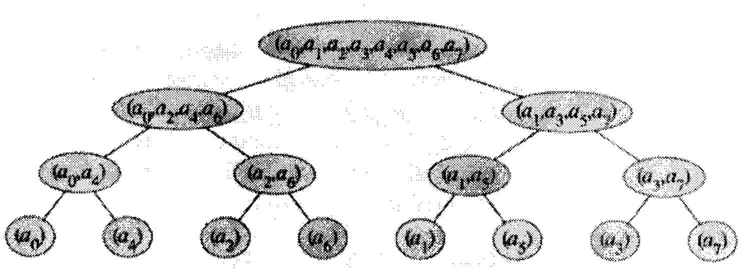


Figure 32.4 The tree of input vectors to the recursive calls of the RECURSIVE-FFT procedure. The initial invocation is for $n = 8$.

An iterative FFT implementation

We first note that the **for** loop of lines 10–13 of RECURSIVE-FFT involves computing the value $\omega_n^k y_k^{[1]}$ twice. In compiler terminology, this value is known as a ***common subexpression***. We can change the loop to compute it only once, storing it in a temporary variable t.

for $k \leftarrow 0$ **to** $n/2 - 1$
  do $t \leftarrow \omega y_k^{[1]}$
    $y_k \leftarrow y_k^{[0]} + t$
    $y_{k+(n/2)} \leftarrow y_k^{[0]} - t$
    $\omega \leftarrow \omega\, \omega_n$

The operation in this loop, multiplying ω (which is equal to ω_n^k) by $y_k^{[1]}$, storing the product into t, and adding and subtracting t from $y_k^{[0]}$, is known as a ***butterfly operation*** and is shown schematically in Figure 32.3.

We now show how to make the FFT algorithm iterative rather than recursive in structure. In Figure 32.4, we have arranged the input vectors to the recursive calls in an invocation of RECURSIVE-FFT in a tree structure, where the initial call is for $n = 8$. The tree has one node for each call of the procedure, labeled by the corresponding input vector. Each RECURSIVE-FFT invocation makes two recursive calls, unless it has received a 1-element vector. We make the first call the left child and the second call the right child.

Looking at the tree, we observe that if we could arrange the elements of the initial vector a into the order in which they appear in the leaves, we could mimic the execution of the RECURSIVE-FFT procedure as follows. First, we take the elements in pairs, compute the DFT of each pair using one butterfly operation, and replace the pair with its DFT. The vector then holds $n/2$ 2-element DFT's. Next, we take these $n/2$ DFT's in pairs and compute the DFT of the four vector elements they come from by executing two butterfly operations, replacing two 2-element DFT's with one 4-element DFT. The vector then holds $n/4$ 4-element DFT's. We continue in this manner until the vector holds two $(n/2)$-element DFT's, which we can combine using $n/2$ butterfly operations into the final n-element DFT.

To turn this observation into code, we use an array $A[0..n-1]$ that initially holds the elements of the input vector a in the order in which they appear in the leaves of the tree of Figure 32.4. (We shall show later how to determine this order.) Because the combining has to be done on each level of the tree, we introduce a variable s to count the levels, ranging from 1 (at the bottom, when we are combining pairs to form 2-element DFT's) to $\lg n$ (at the top, when we are combining two $(n/2)$-element DFT's to produce the final result). The algorithm therefore has the following structure:

```
1  for s ← 1 to lg n
2       do for k ← 0 to n − 1 by 2^s
3              do combine the two 2^(s−1)-element DFT's in
                      A[k .. k + 2^(s−1) − 1] and A[k + 2^(s−1) .. k + 2^s − 1]
                      into one 2^s-element DFT in A[k .. k + 2^s − 1]
```

We can express the body of the loop (line 3) as more precise pseudocode. We copy the **for** loop from the RECURSIVE-FFT procedure, identifying $y^{[0]}$ with $A[k..k+2^{s-1}-1]$ and $y^{[1]}$ with $A[k+2^{s-1}..k+2^s-1]$. The value of ω used in each butterfly operation depends on the value of s; we use ω_m, where $m = 2^s$. (We introduce the variable m solely for the sake of readability.) We introduce another temporary variable u that allows us to perform the butterfly operation in place. When we replace line 3 of the overall structure by the loop body, we get the following pseudocode, which forms the basis of our final iterative FFT algorithm as well as the parallel implementation we shall present later.

FFT-BASE(a)

```
n ← length[a]                ▷ n is a power of 2.
for s ← 1 to lg n
     do m ← 2^s
        ω_m ← e^{2πi/m}
        for k ← 0 to n − 1 by m
            do ω ← 1
               for j ← 0 to m/2 − 1
                   do t ← ω A[k + j + m/2]
                      u ← A[k + j]
                      A[k + j] ← u + t
                      A[k + j + m/2] ← u − t
                      ω ← ω ω_m
```

We now present the final version of our iterative FFT code, which inverts the two inner loops to eliminate some index computation and uses the auxiliary procedure BIT-REVERSE-COPY(a, A) to copy vector a into array A in the initial order in which we need the values.

ITERATIVE-FFT(a)

```
BIT-REVERSE-COPY(a, A)
n ← length[a]                ▷ n is a power of 2.
for s ← 1 to lg n
     do m ← 2^s
        ω_m ← e^{2πi/m}
        ω ← 1
        for j ← 0 to m/2 − 1
            do for k ← j to n − 1 by m
                   do t ← ω A[k + m/2]
                      u ← A[k]
                      A[k] ← u + t
                      A[k + m/2] ← u − t
               ω ← ω ω_m
return A
```

How does BIT-REVERSE-COPY get the elements of the input vector a into the desired order in the array A? The order in which the leaves appear in Figure 32.4 is "bit-reverse binary." That is, if we let $\text{rev}(k)$ be the $\lg n$-bit integer formed by reversing the bits of the binary representation of k, then we want to place vector element a_k in array position $A[\text{rev}(k)]$. In Figure 32.4, for example, the leaves appear in the order $0, 4, 2, 6, 1, 5, 3, 7$; this sequence in binary is $000, 100, 010, 110, 001, 101, 011, 111$, and in bit-reverse binary we get the sequence $000, 001, 010, 011, 100, 101, 110, 111$. To see that we want bit-reverse binary order in general, we note that at the top level of the tree, indices whose low-order bit is 0 are placed in the left subtree and indices whose low-order bit is 1 are placed in the right subtree.

Stripping off the low-order bit at each level, we continue this process down the tree, until we get the bit-reverse binary order at the leaves.

Since the function $\text{rev}(k)$ is easily computed, the BIT-REVERSE-COPY procedure can be written as follows.

```
BIT-REVERSE-COPY(a, A)
n ← length[a]
for k ← 0 to n − 1
    do A[rev(k)] ← a_k
```

The iterative FFT implementation runs in time $\Theta(n \lg n)$. The call to BIT-REVERSE-COPY(a, A) certainly runs in $O(n \lg n)$ time, since we iterate n times and can reverse an integer between 0 and $n-1$, with $\lg n$ bits, in $O(\lg n)$ time. (In practice, we usually know the initial value of n in advance, so we would probably code a table mapping k to $\text{rev}(k)$, making BIT-REVERSE-COPY run in $\Theta(n)$ time with a low hidden constant. Alternatively, we could use the clever amortized reverse binary counter scheme described in Problem 18-1.) To complete the proof that ITERATIVE-FFT runs in time $\Theta(n \lg n)$, we show that $L(n)$, the number of times the body of the innermost loop (lines 9–12) is executed, is $\Theta(n \lg n)$. We have

$$\begin{aligned} L(n) &= \sum_{s=1}^{\lg n} \sum_{j=0}^{2^{s-1}-1} \frac{n}{2^s} \\ &= \sum_{s=1}^{\lg n} \frac{n}{2^s} \cdot 2^{s-1} \\ &= \sum_{s=1}^{\lg n} \frac{n}{2} \\ &= \Theta(n \lg n) . \end{aligned}$$

A parallel FFT circuit

We can exploit many of the properties that allowed us to implement an efficient iterative FFT algorithm to produce an efficient parallel algorithm for the FFT. (See Chapter 29 for a description of the combinational-circuit model.) The combinational circuit PARALLEL-FFT that computes the FFT on n inputs is shown in Figure 32.5 for $n = 8$. The circuit begins with a bit-reverse permutation of the inputs, followed by $\lg n$ stages, each stage consisting of $n/2$ butterflies executed in parallel. The depth of the circuit is therefore $\Theta(\lg n)$.

The leftmost part of the circuit PARALLEL-FFT performs the bit-reverse permutation, and the remainder mimics the iterative FFT-BASE procedure. We take advantage of the fact that each iteration of the outermost **for** loop performs $n/2$ independent butterfly operations that can be per-

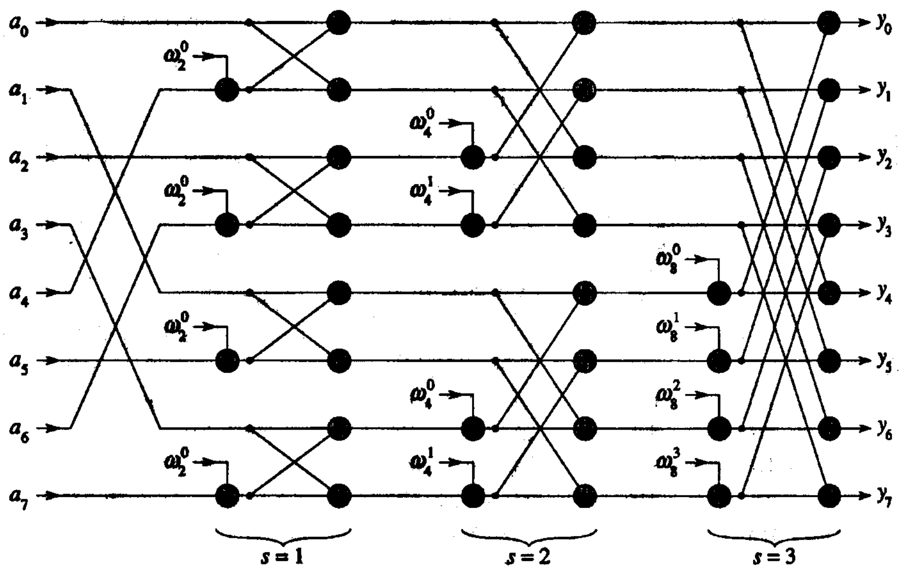


Figure 32.5 A combinational circuit PARALLEL-FFT that computes the FFT, here shown on $n = 8$ inputs. The stages of butterflies are labeled to correspond to iterations of the outermost loop of the FFT-BASE procedure. An FFT on n inputs can be computed in $\Theta(\lg n)$ depth with $\Theta(n \lg n)$ combinational elements.

formed in parallel. The value of s in each iteration within FFT-BASE corresponds to a stage of butterflies shown in Figure 32.5. Within stage s, for $s = 1, 2, \ldots, \lg n$, there are $n/2^s$ groups of butterflies (corresponding to each value of k in FFT-BASE), with 2^{s-1} butterflies per group (corresponding to each value of j in FFT-BASE). The butterflies shown in Figure 32.5 correspond to the butterfly operations of the innermost loop (lines 8–11 of FFT-BASE). Note also that the values of ω used in the butterflies correspond to those used in FFT-BASE: in stage s, we use $\omega_m^0, \omega_m^1, \ldots, \omega_m^{m/2-1}$, where $m = 2^s$.

Exercises

32.3-1
Show how ITERATIVE-FFT computes the DFT of the input vector $(0, 2, 3, -1, 4, 5, 7, 9)$.

32.3-2
Show how to implement an FFT algorithm with the bit-reversal permutation occurring at the end, rather than at the beginning, of the computation. (*Hint:* Consider the inverse DFT.)

32.3-3
To compute DFT_n, how many addition, subtraction, and multiplication elements, and how many wires, are needed in the PARALLEL-FFT circuit described in this section? (Assume that only one wire is needed to carry a number from one place to another.)

32.3-4 ★
Suppose that the adders in the FFT circuit sometimes fail in such a manner that they always produce a zero output, independent of their inputs. Suppose that exactly one adder has failed, but that you don't know which one. Describe how you can identify the failed adder by supplying inputs to the overall FFT circuit and observing the outputs. Try to make your procedure efficient.

Problems

32-1 Divide-and-conquer multiplication

a. Show how to multiply two linear polynomials $ax + b$ and $cx + d$ using only three multiplications. (*Hint:* One of the multiplications is $(a + b) \cdot (c + d)$.)

b. Give two divide-and-conquer algorithms for multiplying two polynomials of degree-bound n that run in time $\Theta(n^{\lg 3})$. The first algorithm should divide the input polynomial coefficients into a high half and a low half, and the second algorithm should divide them according to whether their index is odd or even.

c. Show that two n-bit integers can be multiplied in $O(n^{\lg 3})$ steps, where each step operates on at most a constant number of 1-bit values.

32-2 Toeplitz matrices

A ***Toeplitz matrix*** is an $n \times n$ matrix $A = (a_{ij})$ such that $a_{ij} = a_{i-1,j-1}$ for $i = 2, 3, \ldots, n$ and $j = 2, 3, \ldots, n$.

a. Is the sum of two Toeplitz matrices necessarily Toeplitz? What about the product?

b. Describe how to represent a Toeplitz matrix so that two $n \times n$ Toeplitz matrices can be added in $O(n)$ time.

c. Give an $O(n \lg n)$-time algorithm for multiplying an $n \times n$ Toeplitz matrix by a vector of length n. Use your representation from part (b).

d. Give an efficient algorithm for multiplying two $n \times n$ Toeplitz matrices. Analyze its running time.

Konstrukcija algoritma za DFT_n, uz opći $n \in \mathbb{N}$.

Ključni prvi korak u konstrukciji brzog algoritma za $n = 2^m$ je mogućnost rastava polinoma A na dva dijela iste duljine

$$n = 2 \cdot \left(\frac{n}{2}\right).$$

Ako je $n = p$ prost broj, onda takva mogućnost ne postoji. Tada je

$$y_k = A(\omega_p^k) = \sum_{j=0}^{p-1} a_j \, \omega_p^{kj}, \quad k = 0, \ldots, p-1.$$

Znamo da je

$$\omega_p^{k \cdot j} = \omega_p^{k \cdot j \bmod p},$$

ali za prost broj p, eksponenti $k \cdot j \bmod p$ za $j = 0, \ldots, p-1$ prolaze svih p vrijednosti

$$0, 1, \ldots, p-1$$

za svaki $k \neq 0$. Naime, ω_p^0 je neutral u multiplikativnoj grupi p-tih korijena iz jedinice, a svih ostalih $p-1$ elemenata su generatori te grupe $\Leftrightarrow$ njihovim potencijama možemo dobiti sve elemente grupe.

Dakle, jedini direktni račun vektora y ide:

(a) korištenjem Hornerove sheme za polinom A stupnja $p-1$ (reda p), što treba

$$p-1 \text{ M}, \ p-1 \text{ A} \quad (\text{nad } \mathbb{C})$$

po svakoj točki ω_p^k (osim za ω_p^0, gdje množenja možemo ignorirati, ali moguću uštedu za jednu točku smo mogli napraviti i ranije, pa nismo ju uzeli u obzir — ionako je nižeg reda veličine od ukupne složenosti).

(b) mogli bismo unaprijed spremiti (kao i prije) cijelu tablicu potencija

$$\omega_p^0, \ldots, \omega_p^{p-1}$$

pa imamo skalarni produkt vektora duljine p po svakoj točki - što je istih

$$p-1\ M,\ p-1\ A$$

operacija nad $\mathbb{C}$.

Dakle, u oba slučaja, za svih p točaka imamo

$$p(p-1)\ M,\ p(p-1)\ A \quad \text{za } DFT_p(a).$$

(množenja bismo mogli smanjiti na $(p-1)\cdot(p-1)$, ali zanemarimo to.).

- Finalno skaliranje za $DFT_p^{-1}(a)$ isto nećemo ovdje brojati!

Vidimo da ako je $n=p$ prost broj, onda ne postoji direktna brza diskretna Fourierova transformacija za a.

(Kasnije ćemo pokazati da se ipak i $DFT_p(a)$ može naći brzo, u $O(p \log p) M$, ali ne direktno, nego transformacijom vektora a i korištenjem konvolucije - kao kod brzog množenja polinoma).

- Pretpostavimo sad da je n složen broj

$$n = n_1 \cdot n_2 \quad , \quad n_1, n_2 > 1.$$

Ovdje su n_1 i n_2 bilo koji faktori, ne nužno prosti brojevi!

Tada A možemo rastaviti u n_1 "blokova", svaki duljine n_2.

Indeksima [∅], [1] za $n_1 = 2$, sada opet odgovaraju ostaci modulo n_1, tj.

$$\ell = [\emptyset], [1], \ldots, [n_1 - 1].$$

(ako idemo kao u prvom DFT_{2m} algoritmu)

$j \to \ell,\ k \to m$
$p \to n_1,\ q \to n_2$

Henrici 3

[Odmah rastavi: $j = \ell + m \cdot n_1$, $\ell = j \bmod n_1$
$k = r + s \cdot n_2$, $r = k \bmod n_2$]

N-3

Niz $a^{[\ell]}$ sadrži one elemente a_j od a koji imaju ostatak ℓ mod n_1. Dakle

$$a^{[\ell]} = (a_\ell, a_{\ell+n_1}, \ldots, a_{\ell+(n_2-1)\cdot n_1}), \quad \ell = 0, \ldots, n_1 - 1.$$

Pripadni polinomi su

$$A^{[\ell]}(x) = a_\ell + a_{\ell+n_1} x + \ldots + a_{\ell+(n_2-1)\cdot n_1} \cdot x^{n_2-1}.$$

Očito treba uvrstiti x^{n_1} i napraviti pravu linearnu kombinaciju:

$$A(x) = \sum_{j=0}^{n-1} a_j x^j = \left\{ j = \ell + m \cdot n_1, \quad \begin{matrix} \ell = 0, \ldots, n_1 - 1 \\ m = 0, \ldots, n_2 - 1 \end{matrix} \right\}$$

$$= \sum_{m=0}^{n_2-1} \sum_{\ell=0}^{n_1-1} a_{\ell+m\cdot n_2} x^{\ell+m\cdot n_1}$$

$$= \sum_{\ell=0}^{n_1-1} x^\ell \cdot \sum_{m=0}^{n_2-1} a_{\ell+m\cdot n_2} (x^{n_1})^m$$

$$= \sum_{\ell=0}^{n_1-1} x^\ell \cdot A^{[\ell]}(x^{n_1}).$$

Sad uvrstimo $x = \omega_n^k$ i iskoristimo

$$\omega_{n_1 \cdot n_2}^{k \cdot n_1} = \omega_{n_2}^k \qquad \text{(za sve } k = 0, \ldots, n-1)$$

onda je

$$y_k = \sum_{\ell=0}^{n_1-1} \omega_n^{\ell \cdot k} \cdot A^{[\ell]}(\omega_n^{k \cdot n_1})$$

→ tu koristimo prethodnu formulu $\omega_{n_1 n_2}^{k \cdot n_1} = \omega_{n_2}^k$

$$= \sum_{\ell=0}^{n_1-1} \omega_n^{\ell \cdot k} \cdot A^{[\ell]}(\omega_{n_2}^k)$$

$= \omega_{n_2}^{k \bmod n_2}$ (iz periodičnosti potencija od ω_{n_2})

$$= \sum_{\ell=0}^{n_1-1} \omega_n^{\ell \cdot k} \cdot A^{[\ell]}(\omega_{n_2}^{k \bmod n_2})$$

Neka je $y^{[\ell]} = DFT_{n_2}(a^{[\ell]})$, $\ell = 0, \dots, n_1 - 1$

To znači da je

$$y^{[\ell]}_{k \bmod n_2} = A^{[\ell]}(\omega_{n_2}^{k \bmod n_2}), \quad \forall k$$

($k \bmod n_2$ uredno prolazi $0, \dots, n_2 - 1$ i to n_1 puta, kad k ide od $\emptyset$ do $n-1$).

Dakle, dobivamo da je

$$y_k = \sum_{\ell=0}^{n_1-1} \omega_n^{\ell \cdot k} \cdot y^{[\ell]}_{k \bmod n_2}, \quad k = 0, \dots, n-1$$

Ako svaki y_k računamo po ovoj formuli, uz pretpostavku da su $\omega_n^{\ell \cdot k}$ tabelirani, imamo n skalarnih produkata vektora duljine n_2.

Čak malo bolje, jer znamo da je

$$\omega_n^{\ell \cdot k} = 1 \quad \text{za } \ell = 0,$$

pa ova formula ima oblik

$$y_k = y^{[\emptyset]}_{k \bmod n_2} + \sum_{\ell=1}^{n_1-1} \omega_n^{\ell \cdot k} \, y^{[\ell]}_{k \bmod n_2}, \quad k = 0, \dots, n-1$$

dakle, trebamo

$$n_1 - 1 \ M, \quad n_1 - 1 A \quad \text{po svakom } y_k$$

ili ukupno

$$n \cdot (n_1 - 1) \ M, \quad n \cdot (n_1 - 1) \ A \quad (\text{nad } \mathbb{C})$$

operacija da iz vektora $y^{[\ell]}$, $\ell = 0, \dots, n_1 - 1$ izračunamo traženi y.

Vidimo da u ovoj formulaciji imamo isti broj množenja M i zbrajanja A (nad $\mathbb{C}$), pa možemo analizirati samo množenja.

Potencijalno možemo još ponešto uštediti, ako koristimo ostatke modulo n_2 i u računanju

$$\omega_n^{\ell \cdot k}.$$

Napišimo $k = r + s \cdot n_2$, $r = 0, \ldots, n_2 - 1$
$s = 0, \ldots, n_1 - 1$.

Tada je

$$\omega_n^{\ell \cdot k} = \omega_n^{\ell(r + s \cdot n_2)} = \omega_n^{\ell \cdot r} \cdot \omega_{\underbrace{n}_{n_1 \cdot n_2}}^{\ell s \cdot n_2} =$$

$$= \underline{\underline{\omega_n^{\ell \cdot r} \cdot \omega_{n_1}^{\ell \cdot s}}}.$$

(Svaki niz $y^{[\ell]}_{k \bmod n_2} = y^{[\ell]}_r$ ima period n_2 !)

$$y_{r + s \cdot n_2} = y_r^{[\emptyset]} + \sum_{\ell=1}^{n_1 - 1} \omega_n^{\ell \cdot r} \cdot \omega_{n_1}^{\ell \cdot s} \cdot y_r^{[\ell]}, \quad r = , \; s =$$

$\omega_{n_1}^{\ell \cdot s}$ → ovisi samo o n_1

To računamo tako da izračunamo produkte:

(A) $$z_r^{[\ell]} = \omega_n^{\ell \cdot r} \cdot y_r^{[\ell]} \qquad \ell = (1), \ldots, n_1 - 1, \quad r = 0, \ldots, n_2 - 1$$

tj. njih $(n_1 - 1) \cdot n_2$,

a zatim produkte

(B) $$\omega_{n_1}^{\ell \cdot s} \cdot z_r^{[\ell]} \qquad \ell, r \text{ kao gore}, \quad s = (1), \ldots, n_1 - 1$$

tj. njih $(n_1 - 1)^2 \cdot n_2$.

Kad zbrojimo, to je

$$(n_1 - 1) \cdot n_2 + (n_1 - 1)^2 \cdot n_2 = n_1 \cdot n_2 (n_1 - 1)$$

što nije neka ušteda – općenito.

No, za $n_1 = 2$ je $\omega_{n_1} = \omega_2 = -1$, pa se (B) svodi na promjenu znaka, a ne na množenje !

Dakle, umjesto $n(n_1-1) = n\,M$
imamo samo $n_2 \cdot (n_1-1) = \frac{n}{2} M.$

– Vratimo se prvoj relaciji za broj množenja, jer ona vrijedi za bilo koji rastav $n = n_1 \cdot n_2$

$$M(n_1 \cdot n_2) = n \cdot (n_1 - 1) + n_1 \cdot M(n_2) \qquad n_2 = m_1 \cdot m_2$$

$$n_1 \cdot (n_2 \cdot (m_1 - 1) + m_1 \cdot M(m_2))$$

$$= n_1 \cdot (m_1 - 1) + n \cdot (m_1 - 1) + n_1 \cdot m_1 \cdot M(m_2)$$

prod $= n$

– Lako se vidi da za rastav

$$n = n_1 \cdot n_2 \cdot \ldots \cdot n_e$$

vrijedi

$$M(n) = n \cdot \left[(n_1 - 1) + \ldots + (n_e - 1) \right]$$

(konzist. s $M(p) = p(p-1)$)

– Ostaje pitanje kako treba izabrati rastav pa da ovo bude najmanje!

– Pitanje se svodi na to da li se više isplati faktorizirati $n = n_1 \cdot n_2$ ili ostavti Hornera za n?

$$n\left[(n_1-1) + (n_2-1)\right] \overset{?}{\leq} n(n-1)$$

$n_1 \cdot n_2 - 1$

pitanje: $n_1 + n_2 - 2 \overset{?}{\leq} n_1 \cdot n_2 - 1$

$$(n_1 - 1)(n_2 - 1) \overset{?}{\geq} 0$$

a ovo uvijek vrijedi, jer je $n_1, n_2 \geq 1$. Čak, jače, $=$ vrijedi $\Leftrightarrow n_1 = 1$ ili $n_2 = 1$ tj. $n = 1$ ili prost (što već znamo)

Dakle, faktoriziraj dok ide.
To znači – na proste faktore!

za $n \geq 2$: $\quad n = p_1^{\alpha_1} \cdot \ldots \cdot p_e^{\alpha_e}$

$$M(n) = n \cdot \left[\alpha_1 (p_1 - 1) + \ldots + \alpha_e (p_e - 1) \right].$$

a mi već imamo barem $n \cdot \lg n$. Što je bolje!?

Što smo zapravo napravili?

Koristeći rastav $n = n_1 \cdot n_2$, sveli smo računanje diskretne Fourierove transformacije duljine n, tj. DFT_n, na

1. n_1 računanja DFT_{n_2}, da izračunamo:
$$y^{[\ell]} = DFT_{n_2}(a^{[\ell]}), \quad \ell = 0, \ldots, n_1 - 1$$

2. petlju za računanje $y = DFT_n(a)$ iz $y^{[\ell]}$ u obliku:
$$y_k = y^{[0]}_{k \bmod n_2} + \sum_{\ell=1}^{n_1-1} \omega_n^{\ell \cdot k} \cdot y^{[\ell]}_{k \bmod n_2}, \quad k = 0, \ldots, n-1.$$

Ova zadnja relacija nije ništa drugo nego Hornerova shema za
$$A(x) = \sum_{\ell=0}^{n_1-1} A^{[\ell]}(x^{n_1}) \cdot x^{\ell}$$
u točkama $x = \omega_n^k$, $k = 0, \ldots, n-1$, ili skalarni produkt vektora duljine n_1, s tim da u prvom sumandu ($\ell = 0$), nema množenja.

Ako s $M(n)$ označimo broj kompleksnih množenja za računanje DFT_n, onda po ovom algoritmu trebamo
$$M(n) = \underbrace{n \cdot (n_1 - 1)}_{\text{faza 2 - Horner}} + \underbrace{n_1 \cdot M(n_2)}_{\text{faza 1 - } n_1 \times DFT_{n_2}},$$
s tim da je $n = n_1 \cdot n_2$.

Prvo uočimo da algoritam i prethodna formula vrijede i za slučaj da je $n = p$ prost broj. Tada znamo da je
$$(\min)\ M(p) = p \cdot (p-1).$$
No, znamo i to da je
$$(\min)\ M(1) = 0,$$
jer se računanje $y = DFT_1(a)$ svodi na kopiranje $y = a$ (ili $y_0 = a_0$), pa nema aritmetičkih operacija.

Ako $n=p$ "faktoriziramo" u obliku $p = 1 \cdot p$, tj. uzmemo $n_1 = 1$, $n_2 = p$, onda u fazi 1 imamo jedan DFT_p (što je ekvivalentno polaznom problemu), a faza 2 se svodi na n kopiranja $y_k = y_k^{[\emptyset]}$ (jer je $k \bmod n = k \bmod p = k$). Dakle:

$$M(p) = \underbrace{p \cdot (1-1)}_{\emptyset} + 1 \cdot M(p) = M(p)$$

što je besmisleno, ali pokazuje da je relacija za $M(n)$ konzistentna s $n_1 = 1$ – ne samo za $n=p$ već i općenito:

$$n = 1 \cdot n \qquad M(n) = \underbrace{n \cdot (1-1)}_{\emptyset} + 1 \cdot M(n) = M(n).$$

- S druge strane, ako pišemo $p = p \cdot 1$, onda u fazi 1 imamo p "računanja" DFT_1, što je p kopiranja, bez aritmetičkih operacija, a u fazi 2 imamo Hornerovu shemu za polinom reda p (stupnja $p-1$) i to p puta, što bismo i inače iskoristili za DFT_p.

Dakle, u ovim rastavima smijemo uzeti (općenito) $n_2 = 1$, pa se algoritam svodi na običnu Hornerovu shemu

$$n = n \cdot 1 \qquad M(n) = n \cdot (n-1) + n \cdot \underbrace{M(1)}_{=\emptyset} = n \cdot (n-1).$$

- Ostaje, naravno, ključno pitanje: što se više isplati; faktorizirati n i kako, ili koristiti Hornerovu shemu, tj. kako treba faktorizirati n tako da dobijemo

$$\min M(n)$$

gdje min ide po svim faktorizacijama od n.

Prethodni rastav $n = n \cdot 1$ pokazuje da faktore jednake 1 na kraju možemo ignorirati i svesti na Hornerovu shemu.

Neka je $H(n) = n \cdot (n-1)$ funkcija koja opisuje broj množenja u Hornerovoj shemi za DFT_n.

– Pokazali smo da za bilo koji rastav $n = n_1 \cdot n_2$, $n_1, n_2 \geq 1$, vrijedi

$$M(n) = M(n_1 \cdot n_2) = n \cdot (n_1 - 1) + n_1 \cdot M(n_2).$$

Kad bismo za zadnji faktor n_2, za računanje DFT_{n_2} iskoristili Hornerovu shemu (što moramo, ako je n_2 prost),

$$M(n_2) = H(n_2) = n_2 \cdot (n_2 - 1)$$

dobili bismo

$$M(n) = n \cdot (n_1 - 1) + n_1 \cdot H(n_2) = n \cdot (n_1 - 1) + \underbrace{n_1 \cdot n_2}_{= n} (n_2 - 1)$$
$$= n \cdot [(n_1 - 1) + (n_2 - 1)].$$

Kada je to bolje od obične Hornerove sheme $H(n)$? Izračunajmo razliku $H(n) - M(n)$:

$$H(n) - M(n) = n \cdot (\underset{\uparrow \; = n_1 \cdot n_2}{n} - 1) - n \cdot [(n_1 - 1) + (n_2 - 1)]$$
$$= n \cdot [n_1 \cdot n_2 - 1 - n_1 + 1 - n_2 + 1]$$
$$= n \cdot [n_1 \cdot n_2 - n_1 - n_2 + 1]$$
$$= n \cdot (n_1 - 1)(n_2 - 1)$$

Znamo da je $n, n_1, n_2 \geq 1$, pa je desna strana sigurno nenegativna, tj. vrijedi

$$H(n) - M(n) = n \cdot (n_1 - 1)(n_2 - 1) \geq 0$$

ili

$$\underline{M(n) \leq H(n)}$$

za svaki $n \in \mathbb{N}$ i za svaki rastav $n = n_1 \cdot n_2$, $n_1, n_2 \geq 1$. ($n_1, n_2 \in \mathbb{N}$).

Odmah vidimo da se jednakost $M(n) = H(n)$ dostiže ako i samo ako je

$$n_1 = 1 \text{ ili } n_2 = 1.$$

Kad to primijenimo na naš rekurzivni "algoritam" (ili pristup) za računanje DFT_n dobivamo sljedeći zaključak.

Ako je $n \in \mathbb{N}$ složen broj, onda bilo kojom faktorizacijom

$$n = n_1 \cdot n_2 \,, \quad n_1, n_2 > 1 \,,$$

dobivamo rekurzivni algoritam za računanje DFT_n koji je brži (tj. ima manje kompleksnih aritmetičkih operacija) od Hornerove sheme za DFT_n.

Ovakvo rekurzivno ubrzanje nije moguće ako i samo ako je $n = 1$ ili $n = p$ prost broj.

- Ostaje još pitanje kako treba rastaviti (složeni) zadani broj n da dobijemo najmanji mogući broj množenja $M(n)$. Odgovor na to pitanje kaže kako treba organizirati nivoe rekurzije u rekurzivnom DFT_n algoritmu, tako da dobijemo najbrži mogući algoritam (tzv. Fast Discrete Fourier Transform, ili FFT_n).

Pretpostavimo da smo $n \in \mathbb{N}$ rastavili na faktore u obliku

$$n = n_1 \cdot n_2 \cdot \ldots \cdot n_g$$

gdje je $g \in \mathbb{N}$ i $n_1, \ldots, n_g \geq 1$. Lako se vidi da broj množenja u pripadnom rekurzivnom DFT_n algoritmu vrijedi

$$M(n) = n \cdot \left[(n_1 - 1) + (n_2 - 1) + \ldots + (n_g - 1) \right]$$

i to bez obzira na "poredak" faktora, odnosno način realizacije rekurzije, sve dok za najdublji nivo koristimo Hornerovu shemu.

Jedan od načina organizacije rekurzije je redom

$$DFT_n \to \underbrace{DFT_{n_2 \cdot \ldots \cdot n_g}}_{n_1 \times} \to \underbrace{DFT_{n_3 \cdot \ldots \cdot n_g}}_{(n_1 \times) n_2 \times} \to \ldots \to \underbrace{DFT_{n_g}}_{(n_1 \times \ldots) \times n_{g-1} \times}$$

Treba naći najmanju vrijednost $\min M(n)$, po svim takvim rastavima $n = n_1 \cdot \ldots \cdot n_g$, za sve g. Označimo

$$M^*(n) = \min_{\substack{n = n_1 \cdot \ldots \cdot n_g \\ \text{i po } g}} M(n)$$

Vidimo da $M(n)$ uvijek ima faktor n, pa označimo

$$M(n) = n \cdot m(n) \ , \quad m(n) = (n_1 - 1) + \ldots + (n_g - 1)$$

za dani rastav $n = n_1 \cdot \ldots \cdot n_g$. Treba naći

$$m^*(n) = \min m(n)$$

po svim rastavima od n na faktore, jer očito vrijedi

$$M^*(n) = n \cdot m^*(n).$$

- Znamo već da rekurzivni pristup nema prednosti pred Hornerovom shemom, ako (i samo ako) je $n=1$ ili $n=p$ prost. Dakle, znamo:

$$m^*(n) = \begin{cases} \emptyset & , \text{ za } n=1 \\ p-1 & , \text{ za } n=p \text{ prost.} \end{cases}$$

- Već smo dokazali da za $n = n_1 \cdot n_2$, uz $n_1, n_2 > 1$, vrijedi

$$M(n) < H(n)$$

ili

$$m(n) = (n_1 - 1) + (n_2 - 1) < n_1 \cdot n_2 - 1 = n - 1.$$

To sugerira (čak diktira) svaki složeni faktor treba rastavljati dok god to možemo, a to znači na proste faktore.

Dakle, tvrdimo da se $m^*(n)$ postiže na rastavu od n na proste faktore, za svaki $n \geq 2$. (Naime, $n=1$ po definiciji nije prost, a $m^*(1) = 0$ ionako znamo!)

Po istom principu, pretpostavimo da je neki faktor n_i složen, u rastavu $n = n_1 \cdot \ldots \cdot n_g$, za neki $i \in \{1, \ldots, g\}$ (Naravno, tada je i n složen). Ovom rastavu odgovara

$$m(n_1 \cdot \ldots \cdot n_i \cdot \ldots \cdot n_g) = (n_1 - 1) + \ldots + (n_i - 1) + \ldots + (n_g - 1).$$

Rastavimo li n_i na netrivijalne faktore $n_i = n_{i,1} \cdot n_{i,2}$, uz $n_{i,1}, n_{i,2} > 1$, onda je

$$m(n_1 \cdot \ldots \cdot n_{i,1} \cdot n_{i,2} \cdot \ldots \cdot n_g) = (n_1 - 1) + \ldots + (n_{i,1} - 1) + (n_{i,2} - 1) + \ldots + (n_g - 1)$$

No, zbog

$$(n_{i,1} - 1)(n_{i,2} - 1) > 0$$

$$\Rightarrow \quad (n_{i,1} - 1) + (n_{i,2} - 1) < n_{i,1} \cdot n_{i,2} = n_i$$

pa faktorizacijom dobivamo manju vrijednost funkcije m

$$m(n_1 \cdot \ldots \cdot n_{i,1} \cdot n_{i,2} \cdot \ldots \cdot n_g) < m(n_1 \cdot \ldots \cdot n_i \cdot \ldots \cdot n_g).$$

Odavde odmah slijedi da se $m^*(n)$ dostiže na rastavu od n koji ima samo <u>proste</u> faktore. U protivnom, rastavom bilo kojeg složenog faktora dobivamo manju vrijednost funkcije.

Znamo da svaki prirodni broj $n \geq 2$ možemo, i to jednoznačno, rastaviti u produkt prostih faktora

$$n = p_1^{\alpha_1} \cdot \ldots \cdot p_t^{\alpha_t}$$

gdje su $p_1 < \ldots < p_t$ prosti brojevi i $\alpha_1, \ldots, \alpha_t > 0$ prirodni eksponenti ($p_i^{\alpha_i} = 1$ za $\alpha_i = 0$, ionako ne igra ulogu). Također, znamo da poredak faktora za rekurzivni DFT_n nije bitan i $m(n)$, pa onda i $m^*(n)$, ne ovise o poretku faktora (komutativnost zbroja).

Dakle, za $n \geq 2$ vrijedi

$$m^*(n) = \alpha_1(p_1 - 1) + \ldots + \alpha_t(p_t - 1).$$

(Ionako znamo $m^*(1) = 0$).

Drugim riječima, optimalni rekurzivni DFT_n, tj. FFT_n dobivamo kad n rastavimo na proste faktore i <u>tada</u> za broj množenja vrijedi

$$M^*(n) = n \cdot [\alpha_1(p_1 - 1) + \ldots + \alpha_t(p_t - 1)].$$

($g = \alpha_1 + \ldots + \alpha_g > 0$).

- Zaključujemo da brza varijanta diskretne Fourierove transformacije FFT_n "postoji" za svaki složeni prirodni broj n (u smislu da je FFT_n brži od Hornerove sheme).

- Na kraju, usporedimo ovaj rezultat s najpopularnijim "klasičnim" izborom za n, a to je kad je n potencija od 2

$$n = 2^m, \quad m \in \mathbb{N}_0.$$

$(t = 1, p_1 = 2, \alpha_1 = m)$.

Tada je: $m^*(2^m) = m \cdot \underbrace{(2-1)}_{1} = m = \lg n$

i: $M^*(n) = M^*(2^m) = n \cdot m = n \cdot \lg n$

(Napomena: ovo odgovara sporijoj varijanti DFT_{2^m}, bez pomoćne varijable i odurimanja. O dodatnim uštedama malo kasnije).

Naravno, $f(n) = n \cdot \lg n$, je kao funkcija, korektno definirana za $\forall n \in \mathbb{N}$, pa možemo uspoređivati $M^*(n)$ i $f(n) = n \cdot \lg n$ za sve n.

Upravo smo vidjeli

$$M^*(n) = n \cdot \lg n = f(n), \quad za\ n = 2^m, m \in \mathbb{N}_0$$

Koji je odnos za ostale n, kad n nije potencija od 2. Treba usporediti $m^*(n)$ i $\lg n$, u rastavu n na proste faktore:

$$n = p_1^{\alpha_1} \cdot \ldots \cdot p_t^{\alpha_t}.$$

Tada je:

$$\lg n = \alpha_1 \cdot \lg p_1 + \ldots + \alpha_t \cdot \lg p_t$$

i

$$m^*(n) = \alpha_1 \cdot (p_1 - 1) + \ldots + \alpha_t (p_t - 1).$$

No, odmah vidimo da je

$$\lg n \leq n - 1$$

za svaki $n \in \mathbb{N}$, s tim da se jednakost dostiže za $n = 1$ i $n = 2$. Osim $p_1 = 2$, svi ostali prosti brojevi su veći od 2

$$t \geq 2 \Rightarrow p_t > 2 \Rightarrow \lg p_t < p_t - 1$$

pa iz $\alpha_t > 0 \Rightarrow$

$$m^*(n) < \lg n$$

čim n ima faktor $p_t > 2$.

U tom smislu, zaključujemo da "najbrži" mogući FFT_n dobivamo kad je n potencija od 2. Upravo zato se $n = 2^m$ koristi kad god je to moguće.

Nažalost, to ne ide uvijek. Naime $A(x)$ ili vektor a je lako dopuniti nulama do prve potencije od 2. Međutim, ta promjena n u 2^m mijenja i točke ω_n^k u kojima se računa DFT, pa ne dobivamo iste vrijednosti! U nekim primjenama nas to ne smeta (na primjer, za brzo množenje polinoma), ali u drugima su i te točke bitne. Kako tada treba postupiti – malo kasnije. Pokazat ćemo da se, sličnim putem kao kod brzog množenja polinoma, putem konvolucije, može za bilo koji n dobiti DFT_n u $O(n \lg n)$ kompleksnih množenja.

Na kraju, odgovorimo na pitanje da li se u općem rekurzivnom DFT_n algoritmu može napraviti slična ušteda polovine množenja kao i u DFT_{2^m}.

Druga faza rekurzivnog algoritma (nakon rekurzivnih poziva) je računanje

$$y_k = y^{[\emptyset]}_{k \bmod n_2} + \sum_{l=1}^{n_1-1} \omega_n^{l\cdot k} \cdot y^{[e]}_{k \bmod n_2}, \quad k = 0, \ldots, n-1.$$

Uštedu treba napraviti, ako iole, koristeći neke pravilnosti i relacije za

$$\omega_n^{l\cdot k}$$

vezane uz ostatke modulo n_2. (l je već $= j \bmod n_1$).

Napišimo stoga k u obliku

$$k = r + s\cdot n_2, \quad \begin{cases} r = 0, \ldots, n_2 - 1 \\ s = 0, \ldots, n_1 - 1, \end{cases}$$

pa je $k \bmod n_2 = r$. Onda je

$$\omega_n^{l\cdot k} = \omega_n^{l\cdot(r + s\cdot n_2)} = \omega_n^{l\cdot r} \cdot \omega_{\underbrace{n}_{n_1\cdot n_2}}^{l\cdot s\cdot n_2} = \omega_n^{l\cdot r} \cdot \omega_{n_1}^{l\cdot s}$$

pa je:

$$y_{r+s\cdot n_2} = y_r^{[\emptyset]} + \sum_{l=1}^{n_1-1} \underbrace{\omega_{n_1}^{l\cdot s}}_{\substack{\text{ovisi samo} \\ \text{o } n_1 \\ \text{(potencijalna ušteda)}}} \cdot \omega_n^{l\cdot r} \cdot y_r^{[e]}, \quad \begin{cases} r = 0, \ldots, n_2 - 1, \\ s = 0, \ldots, n_1 - 1. \end{cases}$$

Ove izraze – produkte računamo u drugoj fazi. (Na zbrajanju, očito, nećemo moći napraviti uštedu – sve sumande će trebati zbrojiti!)

(A): Prvo računamo desne produkte koji ne ovise o s, već samo o r i ℓ:

$$z_r^{[e]} = \omega_n^{\ell \cdot r} \cdot y_r^{[e]}, \quad \begin{cases} \ell = (1), \dots, n_1 - 1 \\ r = 0, \dots, n_2 - 1. \end{cases}$$

Tih produkata ima $(n_1 - 1) \cdot n_2$.

Onog za $\ell = 0$ ionako ne računamo – pripadni član je već ispred sume. Ako izbrojimo sva ova množenja, broj množenja u fazi A je

$$(n_1 - 1) \cdot n_2 \ M.$$

Na produktima za $r = 0$ je $z_0^{[e]} = y_0^{[e]}$, pa bismo mogli uštediti po jedno množenje i ukupni broj množenja bi bio

$$(n_1 - 1)(n_2 - 1) \ M.$$

To bi odgovaralo (v. malo niže) uštedi svih množenja u računanju y_0, tj. za $k = 0$, a tu (očitu) uštedu ni ranije nismo brojali (smanjili $M(n)$), pa nećemo ni sada inzistirati na njoj.

(B): Zatim izračunamo produkte

$$z_{r,s}^{[e]} = \omega_{n_1}^{\ell \cdot s} \cdot z_r^{[e]}, \quad \begin{cases} \ell, r \text{ kao gore u (A)} \\ s = (1), \dots, n_1 - 1, \end{cases}$$

s tim da koristimo da je $\omega_{n_1}^{\ell \cdot s} = 1$ za $s = 0$, i definiramo:

$$z_{r,0}^{[e]} = z_r^{[e]}.$$

Produkata $z_{r,s}^{[e]}$, $s \neq 0$ ima $(n_1 - 1)^2 \cdot n_2$.

(C): Na kraju, za dovršenje druge faze, treba izračunati još i sume

$$y_{r + s \cdot n_2} = y_r^{[0]} + \sum_{\ell=1}^{n_1 - 1} z_{r,s}^{[e]}, \quad \begin{cases} r = 0, \dots, n_2 - 1 \\ s = 0, \dots, n_1 - 1. \end{cases}$$

(tu više nema množenja).

Ako produkte u fazi B izračunamo tako da zaista <u>množimo</u> sve navedene vrijednosti (faktore), onda imamo

$$(n_1-1)^2 \, n_2 \, M$$

množenja u fazi B. Ovdje ne pomaže ni $r=\emptyset$, jer je $z_0^{[e]} = y_0^{[e]}$, a to može biti bilo što, pa pripadnih množenja ima.

U tom slučaju, ukupno bismo za obje faze A i B zajedno imali sljedeći broj kompleksnih množenja:

$$\underbrace{(n_1-1)\cdot n_2}_{\text{faza A}} + \underbrace{(n_1-1)^2 n_2}_{\text{faza B}} = (n_1-1)\cdot n_2 \cdot [\underbrace{1+(n_1-1)}_{n_1}]$$

$$= n_1 \cdot n_2 \cdot (n_1-1) = n\cdot(n_1-1),$$

a to je isto kao i ranije u Hornerovoj shemi u fazi 2.

Čak i kad bismo za fazu A uzeli "pedantni" broj od $(n_1-1)(n_2-1)$ množenja, još uvijek dobivamo:

$$(n_1-1)(n_2-1) + (n_1-1)^2 n_2 = (n_1-1)\cdot[n_2-1+(n_1-1)\cdot n_2]$$

$$= (n_1-1)\cdot[\cancel{n_2}-1+n_1\cdot n_2-\cancel{n_2}]$$

$$= (n_1\cdot n_2-1)\cdot(n_1-1)$$

$$= (n-1)\cdot(n_1-1).$$

Ovo točno odgovara ranijoj potencijalnoj uštedi od n_1-1 množenja u računanju y_0, za $k=0$ u Hornerovoj shemi. Dakle, ne dobivamo ništa bitno novo i bolje.

Gdje je onda ušteda? Još uvijek nije vidljiva!

Ideja je pametno kombinirati faze B i C, tako da ne moramo izračunati baš sve produkte u fazi B, već pokušavamo neke izraziti preko drugih i to uvrstiti u C.

- Pogledajmo prvo situaciju za $n_1 = 2$, neovisno o n_2, tj. nije bitno da n bude potencija od 2, već samo $n = 2 \cdot n_2$ (prvi prosti faktor je 2).

Tada je očito
$$\omega_{n_1} = \omega_2 = -1$$
pa je blesavo množiti s potencijama od -1, kad to možemo realizirati oduzimanjem u fazi C, ako je eksponent baš negativan.

U fazi B je tada $\ell = 1$ (drugih ℓ-ova nema, jer je $n_1 - 1 = 1$, a član za $\ell = 0$ je već na početku sume u fazi C). Isto vrijedi i za s – kod "produktnih" članova – imamo samo
$$z_{r,1}^{[1]} = \omega_2^{1\cdot 1} z_r^{[1]} = -z_r^{[1]}$$
dok je
$$z_{r,0}^{[1]} = z_r^{[1]}$$
po definiciji.

Dakle, množenja u fazi B uopće nisu potrebna ako fazu C realiziramo u paru $s = 0, 1$, kao

$$\left.\begin{aligned} (s=\emptyset):&\quad y_r = y_r^{[\emptyset]} + z_{r,0}^{[1]} = y_r^{[\emptyset]} + z_r^{[1]} \\ (s=1):&\quad y_{r+n_2} = y_r^{[\emptyset]} + z_{r,1}^{[1]} = y_r^{[\emptyset]} - z_r^{[1]} \end{aligned}\right\} \quad r = 0, \ldots, n_2 - 1.$$

Imamo samo $(n_1 - 1)\cdot n_2 = \frac{n}{2}$ množenja u fazi A i štedimo $(n_1-1)^2 n_2 = \frac{n}{2}$ množenja u fazi B.

Štedimo <u>polovinu</u> množenja pri prijelazu iz $DFT_{n/2}$ na DFT_n, čim je $n_1 = 2$. Dakle
$$M(2\cdot n_2) = \frac{n}{2} + 2\cdot M(n_2)$$
a ne $n + 2M(n_2)$. Naravno, to vrijedi za svaki faktor $p_1 = 2$ u rastavu od n na proste faktore, tj. za
$$n = 2^{\alpha_1} \cdot p_i^{\alpha_i} \cdots p_t^{\alpha_t}$$
je broj kompleksnih množenja jednak
$$M'(n) = \frac{1}{2} n\cdot\alpha_1 + n\cdot\left[\alpha_i(p_i - 1) + \ldots + \alpha_t(p_t - 1)\right)$$
$$= \frac{1}{2} n\cdot\alpha_1 + M\left(\frac{n}{2^{\alpha_1}}\right).$$
Broj zbrajanja ostaje isti kao i prije $M(n) = n\cdot\alpha_1 + M\left(\frac{n}{2^{\alpha_1}}\right)$.

Ako je n potencija od 2, tj. $n = 2^m$, $m \in \mathbb{N}_0$, onda je odmah vidljivo da je broj operacija u stvarno brzom algoritmu za DFT_{2^m} jednak

$$\frac{1}{2} n \cdot \lg n \quad \text{kompleksnih množenja}$$

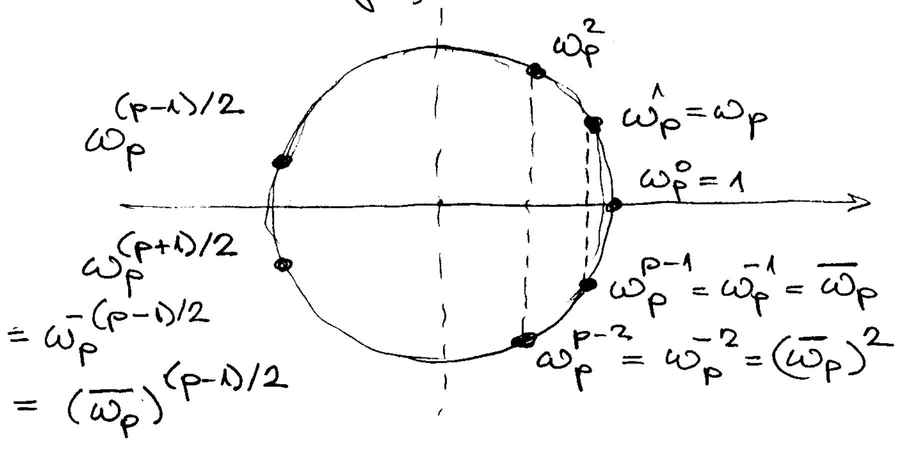
$$n \cdot \lg n \quad \text{kompleksnih zbrajanja.}$$

- Pitanje je da li nešto slično možemo napraviti i kad je $n_1 > 2$.

Prvo uočimo da je, zapravo, dovoljno gledati slučaj kad je $n_1 = p > 2$ neparan prost broj, tj. FFT_n (dakle, najbrži oblik algoritma za DFT_n), uz $n = p \cdot n_2$.

[Ovo nećemo bitno iskoristiti u nastavku, sve ide za bilo koji $n_1 > 2$.]

Tada su potencije ω_p^q simetrično raspoređene po jediničnom krugu, obzirom na realnu os



[Ovo je izgled za neparne p. Isto imam za neparne n_1, a za parne n_1 je $\omega_{n_1}^{n_1/2} = (-1)$

pa taj nema svoj par, ali njega spojimo s $\omega_{n_1}^0 = 1$, kao za $n_1 = 2$].

Obzirom na to da suma ide po ℓ, treba iskoristiti simetriju po s. Vidimo da treba gledati s i $p_1 - s$, s tim da s ide od 1 do $(p_1 - 1)/2$, za $n_1 = p$ neparan (prost).

Nažalost, tada je $\omega_p^q \neq \pm 1$ za $q = 1, \ldots, p-1$ pa simetriju ne možemo iskoristiti na nivou kompleksne aritmetike (nema ni simetrije obzirom na imaginarnu os!! – dokažite).

Pogledajmo onda realizaciju kompleksnih aritmetičkih operacija preko realnih aritmetičkih operacija.

Općenito je za jedno kompleksno množenje potrebno 4 realna množenja i 2 realna zbrajanja

$$1M = 4M_R + 2A_R$$

iz

$$(e + fi) = (a + bi)\cdot(c + di) = = (ac - bd) + i(ad + bc)$$

tj.

$$e = ac - bd \qquad 2M_R + 1A_R$$
$$f = ad + bc \qquad \text{— '' —}.$$

U fazi B gledamo par produkata

$$z_{r,s}^{[e]} = \omega_p^{e\cdot s} \cdot z_r^{[e]}$$

$$z_{r,p-s}^{[e]} = \omega_p^{e\cdot(p-s)} \cdot z_r^{[e]} = \underbrace{\omega_p^{e\cdot p}}_{=1} \cdot \underbrace{\omega_p^{-e\cdot s}}_{(\overline{\omega_p})^{e\cdot s}} \cdot z_r^{[e]}$$

$$= (\overline{\omega_p})^{e\cdot s} \cdot z_r^{[e]} = \overline{(\omega_p^{e\cdot s})} \cdot z_r^{[e]}$$

za $s = 1, \ldots, (p-1)/2$. Direktni račun bez ušteda treba 2M za ove produkte (i još 2A za pripadna zbrajanja u fazi C), tj.

$$2M = 8M_R + 4A_R$$
$$(+\,2A = \qquad 4A_R)$$

No, faktori $\omega_p^{e\cdot s}$ i $\overline{(\omega_p^{e\cdot s})}$ imaju iste realne i suprotne imaginarne dijelove. Usporedbom relacija

$$(a+bi)(c+di) = (ac-bd) + (ad+bc)i$$
$$(a-bi)(c+di) = (ac+bd) + (ad-bc)i$$

vidimo odmah da trebamo samo 4, a ne 8, realnih množenja (i 4 realna zbrajanja ili oduzimanja).

Dakle, za par

$$z_{r,s}^{[e]},\ z_{r,p-s}^{[e]}, \qquad \begin{array}{l} e = (1), \ldots, n_1 - 1 \ \ (p-1) \\ r = 0, \ldots, n_2 - 1 \\ s = (1), \ldots, \frac{n_1 - 1}{2} \ \ \left(\frac{p-1}{2}\right) \end{array}$$

treba $4M_R + 4A_R$ po paru,
što daje uštedu od <u>polovine</u> svih <u>realnih</u> množenja u fazi B (a broj realnih zbrajanja ostaje isti).

Ukupno gledajući, nismo uštedili baš polovinu svih realnih množenja, jer u fazi A nema uštede. Nb, faza A ima za red veličine manje produkata

A: $(p-1)\cdot\frac{n}{p}$ M $\Leftrightarrow$ $4(p-1)\cdot\frac{n}{p}\,M_R$

B: ušteda $\Leftrightarrow$ $2(p-1)^2\cdot\frac{n}{p}\,M_R$

↑ umjesto 4

pa je ušteda blizu $\frac{1}{2}$ ukupnog broja realnih množenja.

U prvoj varijanti imali smo ($1M \to 4M_R$)

$$4n\cdot(p-1)\ M_R$$

a sada imamo

$$4(p-1)\cdot\frac{n}{p}+2(p-1)^2\cdot\frac{n}{p}=2n\cdot(p-1)\Big[\underbrace{\frac{2}{p}+\frac{p-1}{p}}_{\frac{p+1}{p}}\Big]$$

$$=\underbrace{\left(1+\frac{1}{p}\right)}_{\text{ovo je "malo" veće od 1.}}\cdot\underbrace{2n(p-1)}_{\text{ovo bi bila polovina ranijeg broja realnih množenja.}}\ M_R.$$

- Pokažite da isti argument vrijedi za bilo koji neparni n_1 i samo treba promijeniti $p \mapsto n_1$ u zadnjem broju množenja.
- Što se dobije za n_1 paran? [Isto što i za $n_1=2$, tj. točno polovina realnih množenja].

- Znamo da kompleksno množenje možemo realizirati i ovako: $(e+fi)=(a+bi)(c+di)$

$$\left.\begin{aligned}t_1&=ac\\t_2&=bd\\t_3&=(a+b)(c+d)\\e&=t_1-t_2\\f&=t_3-t_1-t_2\end{aligned}\right\}$$

što daje $1M = 3M_R + 5A_R$ (ušteda je $1M_R$ na račun $3A_R$).

Što se tada događa za $n_1=2$, $n_1=p_{>2}$, n_1 neparan, n_1 paran?

32 Polynomials and the FFT

The straightforward method of adding two polynomials of degree n takes $\Theta(n)$ time, but the straightforward method of multiplying them takes $\Theta(n^2)$ time. In this chapter, we shall show how the Fast Fourier Transform, or FFT, can reduce the time to multiply polynomials to $\Theta(n \lg n)$.

Polynomials

A ***polynomial*** in the variable x over an algebraic field F is a function $A(x)$ that can be represented as follows:

$$A(x) = \sum_{j=0}^{n-1} a_j x^j .$$

We call n the ***degree-bound*** of the polynomial, and we call the values $a_0, a_1, \ldots, a_{n-1}$ the ***coefficients*** of the polynomial. The coefficients are drawn from the field F, typically the set $\mathbf{C}$ of complex numbers. A polynomial $A(x)$ is said to have ***degree*** k if its highest nonzero coefficient is a_k. The degree of a polynomial of degree-bound n can be any integer between 0 and $n-1$, inclusive. Conversely, a polynomial of degree k is a polynomial of degree-bound n for any $n > k$.

There are a variety of operations we might wish to define for polynomials. For ***polynomial addition***, if $A(x)$ and $B(x)$ are polynomials of degree-bound n, we say that their ***sum*** is a polynomial $C(x)$, also of degree-bound n, such that $C(x) = A(x) + B(x)$ for all x in the underlying field. That is, if

$$A(x) = \sum_{j=0}^{n-1} a_j x^j$$

and

$$B(x) = \sum_{j=0}^{n-1} b_j x^j ,$$

then

$$C(x) = \sum_{j=0}^{n-1} c_j x^j ,$$

where $c_j = a_j + b_j$ for $j = 0, 1, \ldots, n-1$. For example, if $A(x) = 6x^3 + 7x^2 - 10x + 9$ and $B(x) = -2x^3 + 4x - 5$, then $C(x) = 4x^3 + 7x^2 - 6x + 4$.

For ***polynomial multiplication***, if $A(x)$ and $B(x)$ are polynomials of degree-bound n, we say that their ***product*** $C(x)$ is a polynomial of degree-bound $2n - 1$ such that $C(x) = A(x)B(x)$ for all x in the underlying field. You have probably multiplied polynomials before, by multiplying each term in $A(x)$ by each term in $B(x)$ and combining terms with equal powers. For example, we can multiply $A(x) = 6x^3 + 7x^2 - 10x + 9$ and $B(x) = -2x^3 + 4x - 5$ as follows:

$$\begin{array}{rrrrrrr}
 & & & 6x^3 & +\ 7x^2 & -\ 10x & +\ 9 \\
 & & & -\ 2x^3 & & +\ 4x & -\ 5 \\ \hline
 & & & -\ 30x^3 & -\ 35x^2 & +\ 50x & -\ 45 \\
 & & 24x^4 & +\ 28x^3 & -\ 40x^2 & +\ 36x & \\
 -\ 12x^6 & -\ 14x^5 & +\ 20x^4 & -\ 18x^3 & & & \\ \hline
 -\ 12x^6 & -\ 14x^5 & +\ 44x^4 & -\ 20x^3 & -\ 75x^2 & +\ 86x & -\ 45
\end{array}$$

Another way to express the product $C(x)$ is

$$C(x) = \sum_{j=0}^{2n-2} c_j x^j , \tag{32.1}$$

where

$$c_j = \sum_{k=0}^{j} a_k b_{j-k} . \tag{32.2}$$

Note that $\text{degree}(C) = \text{degree}(A) + \text{degree}(B)$, implying

$$\begin{aligned}
\text{degree-bound}(C) &= \text{degree-bound}(A) + \text{degree-bound}(B) - 1 \\
&\le \text{degree-bound}(A) + \text{degree-bound}(B) .
\end{aligned}$$

We shall nevertheless speak of the degree-bound of C as being the sum of the degree-bounds of A and B, since if a polynomial has degree-bound k it also has degree-bound $k + 1$.

Chapter outline

Section 32.1 presents two ways to represent polynomials: the coefficient representation and the point-value representation. The straightforward methods for multiplying polynomials—equations (32.1) and (32.2)—take $\Theta(n^2)$ time when the polynomials are represented in coefficient form, but only $\Theta(n)$ time when they are represented in point-value form. We can, however, multiply polynomials using the coefficient representation in only $\Theta(n \lg n)$ time by converting between the two representations. To see why this works, we must first study complex roots of unity, which we do in Section 32.2. Then, we use the FFT and its inverse, also described in Section 32.2, to perform the conversions. Section 32.3 shows how to implement the FFT quickly in both serial and parallel models.

This chapter uses complex numbers extensively, and the symbol i will be used exclusively to denote $\sqrt{-1}$.

32.1 Representation of polynomials

The coefficient and point-value representations of polynomials are in a sense equivalent; that is, a polynomial in point-value form has a unique counterpart in coefficient form. In this section, we introduce the two representations and show how they can be combined to allow multiplication of two degree-bound n polynomials in $\Theta(n \lg n)$ time.

Coefficient representation

A ***coefficient representation*** of a polynomial $A(x) = \sum_{j=0}^{n-1} a_j x^j$ of degree-bound n is a vector of coefficients $a = (a_0, a_1, \ldots, a_{n-1})$. In matrix equations in this chapter, we shall generally treat vectors as column vectors.

The coefficient representation is convenient for certain operations on polynomials. For example, the operation of ***evaluating*** the polynomial $A(x)$ at a given point x_0 consists of computing the value of $A(x_0)$. Evaluation takes time $\Theta(n)$ using ***Horner's rule***:

$$A(x_0) = a_0 + x_0(a_1 + x_0(a_2 + \cdots + x_0(a_{n-2} + x_0(a_{n-1}))\cdots)) .$$

Similarly, adding two polynomials represented by the coefficient vectors $a = (a_0, a_1, \ldots, a_{n-1})$ and $b = (b_0, b_1, \ldots, b_{n-1})$ takes $\Theta(n)$ time: we just output the coefficient vector $c = (c_0, c_1, \ldots, c_{n-1})$, where $c_j = a_j + b_j$ for $j = 0, 1, \ldots, n-1$.

Now, consider the multiplication of two degree-bound n polynomials $A(x)$ and $B(x)$ represented in coefficient form. If we use the method described by equations (32.1) and (32.2), polynomial multiplication takes time $\Theta(n^2)$, since each coefficient in the vector a must be multiplied by each coefficient in the vector b. The operation of multiplying polynomials in coefficient form seems to be considerably more difficult than that of evaluating a polynomial or adding two polynomials. The resulting coefficient vector c, given by equation (32.2), is also called the ***convolution*** of the input vectors a and b, denoted $c = a \otimes b$. Since multiplying polynomials and computing convolutions are fundamental computational problems of considerable practical importance, this chapter concentrates on efficient algorithms for them.

Point-value representation

A ***point-value representation*** of a polynomial $A(x)$ of degree-bound n is a set of n ***point-value pairs***

$$\{(x_0, y_0), (x_1, y_1), \ldots, (x_{n-1}, y_{n-1})\}$$

such that all of the x_k are distinct and

$$y_k = A(x_k) \tag{32.3}$$

for $k = 0, 1, \ldots, n-1$. A polynomial has many different point-value representations, since any set of n distinct points $x_0, x_1, \ldots, x_{n-1}$ can be used as a basis for the representation.

Computing a point-value representation for a polynomial given in coefficient form is in principle straightforward, since all we have to do is select n distinct points $x_0, x_1, \ldots, x_{n-1}$ and then evaluate $A(x_k)$ for $k = 0, 1, \ldots, n-1$. With Horner's method, this n-point evaluation takes time $\Theta(n^2)$. We shall see later that if we choose the x_k cleverly, this computation can be accelerated to run in time $\Theta(n \lg n)$.

The inverse of evaluation—determining the coefficient form of a polynomial from a point-value representation—is called ***interpolation***. The following theorem shows that interpolation is well defined, assuming that the degree-bound of the interpolating polynomial equals the number of given point-value pairs.

Theorem 32.1 (Uniqueness of an interpolating polynomial)
For any set $\{(x_0, y_0), (x_1, y_1), \ldots, (x_{n-1}, y_{n-1})\}$ of n point-value pairs, there is a unique polynomial $A(x)$ of degree-bound n such that $y_k = A(x_k)$ for $k = 0, 1, \ldots, n-1$.

Proof The proof is based on the existence of the inverse of a certain matrix. Equation (32.3) is equivalent to the matrix equation

$$\begin{pmatrix} 1 & x_0 & x_0^2 & \cdots & x_0^{n-1} \\ 1 & x_1 & x_1^2 & \cdots & x_1^{n-1} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 1 & x_{n-1} & x_{n-1}^2 & \cdots & x_{n-1}^{n-1} \end{pmatrix} \begin{pmatrix} a_0 \\ a_1 \\ \vdots \\ a_{n-1} \end{pmatrix} = \begin{pmatrix} y_0 \\ y_1 \\ \vdots \\ y_{n-1} \end{pmatrix} . \tag{32.4}$$

The matrix on the left is denoted $V(x_0, x_1, \ldots, x_{n-1})$ and is known as a Vandermonde matrix. By Exercise 31.1-10, this matrix has determinant

$$\prod_{j<k} (x_k - x_j) ,$$

and therefore, by Theorem 31.5, it is invertible (that is, nonsingular) if the x_k are distinct. Thus, the coefficients a_j can be solved for uniquely given the point-value representation:

$$a = V(x_0, x_1, \ldots, x_{n-1})^{-1} y .$$

■

The proof of Theorem 32.1 describes an algorithm for interpolation based on solving the set (32.4) of linear equations. Using the LU decomposition algorithms of Chapter 31, we can solve these equations in time $O(n^3)$.

A faster algorithm for n-point interpolation is based on ***Lagrange's formula***:

$$A(x) = \sum_{k=0}^{n-1} y_k \frac{\prod_{j \neq k}(x - x_j)}{\prod_{j \neq k}(x_k - x_j)} . \tag{32.5}$$

You may wish to verify that the right-hand side of equation (32.5) is a polynomial of degree-bound n that satisfies $A(x_k) = y_k$ for all k. Exercise 32.1-4 asks you how to compute the coefficients of A using Lagrange's formula in time $\Theta(n^2)$.

Thus, n-point evaluation and interpolation are well-defined inverse operations that transform between the coefficient representation of a polynomial and a point-value representation.[1] The algorithms described above for these problems take time $\Theta(n^2)$.

The point-value representation is quite convenient for many operations on polynomials. For addition, if $C(x) = A(x) + B(x)$, then $C(x_k) = A(x_k) + B(x_k)$ for any point x_k. More precisely, if we have a point-value representation for A,

$$\{(x_0, y_0), (x_1, y_1), \ldots, (x_{n-1}, y_{n-1})\} ,$$

and for B,

$$\{(x_0, y'_0), (x_1, y'_1), \ldots, (x_{n-1}, y'_{n-1})\}$$

(note that A and B are evaluated at the *same* n points), then a point-value representation for C is

$$\{(x_0, y_0 + y'_0), (x_1, y_1 + y'_1), \ldots, (x_{n-1}, y_{n-1} + y'_{n-1})\} .$$

The time to add two polynomials of degree-bound n in point-value form is thus $\Theta(n)$.

Similarly, the point-value representation is convenient for multiplying polynomials. If $C(x) = A(x)B(x)$, then $C(x_k) = A(x_k)B(x_k)$ for any point x_k, and we can pointwise multiply a point-value representation for A by a point-value representation for B to obtain a point-value representation for C. We must face the problem, however, that the degree-bound of C is the sum of the degree-bounds for A and B. A standard point-value representation for A and B consists of n point-value pairs for each polynomial. Multiplying these together gives us n point-value pairs for C, but

---

[1]Interpolation is a notoriously tricky problem from the point of view of numerical stability. Although the approaches described here are mathematically correct, small differences in the inputs or round-off errors during computation can cause large differences in the result.