DFT a NTT

mišof

1 Prerekvizity z matematiky

1.1 Lagrangeov tvar interpolačného polynómu

Majme body $(x_0, y_0), \ldots, (x_{n-1}, y_{n-1})$, pričom všetky x_i sú navzájom rôzne. Chceme zostrojiť polynóm stupňa ostro menšieho ako n, ktorý cez všetky tieto body prechádza. Uvažujme nasledujúce polynómy:

$$\ell_j(x) := \prod_{\substack{0 \le m < n \\ m \neq j}} \frac{x - x_m}{x_j - x_m}$$

Každé ℓ_j je skutočne polynóm: v čitateli zlomku je súčin lineárnych termov obsahujúcich premennú x, v menovateli súčin konkrétnych nenulových konštánt.

Ľahko sa presvedčíme, že hodnoty $\ell_j(x_j)$ sú všetky rovné 1, a že pre $i \neq j$ je $\ell_j(x_i) = 0$. To je výborné, lebo vďaka tomu teraz z týchto pomocných polynómov ľahko naskladáme hľadaný interpolačný polynóm:

$$L(x) = \sum_{j=0}^{n-1} y_j \ell_j(x)$$

1.2 Interpolačný polynóm je jednoznačne určený

Na interpoláciu sa môžeme dívať ako na sústavu n lineárnych rovníc (pre každý bod jedna rovnica) s n neznámymi (koeficienty pri x^0 až x^{n-1} v hľadanom polynóme). Matica tejto sústavy je tzv. Vandermondova matica, o ktorej sa dá dokázať, že je vždy regulárna – a teda existuje práve jedno riešenie. (Determinant tejto matice je rovný súčinu všetkých rozdielov tvaru $x_j - x_i$ pre $i < j$.)

Iný dôkaz jednoznačnosti interpolačného polynómu vyplýva zo základnej vety algebry, ktorá hovorí, že každý nekonštantný polynóm nad $\mathbb{C}$ má koreň.

Dôsledky tejto vety: Každý nekonštantný polynóm stupňa n nad $\mathbb{C}$ má práve n koreňov. Každý nekonštantný polynóm nad $\mathbb{C}$ vieme zapísať v tvare súčinu koreňových činiteľov – teda v tvare $p(x) = c \cdot \prod_{i=1}^{n}(x - k_i)$, kde $k_1, \ldots, k_n$ sú jednotlivé korene a c je nejaká konštanta.

Čo by sa stalo, keby existovali dva rôzne interpolačné polynómy p_1 a p_2? Ich rozdielom by bol polynóm r, ktorý nie je konštantný (lebo p_1 a p_2 sú rôzne) a je stupňa menšieho ako n (lebo aj p_1 aj p_2 sú stupňa menšieho ako n). Polynóm r by ale mal aspoň n koreňov (všetky hodnoty x_i sú jeho koreňmi), čo je spor.

1.3 Komplexné odmocniny z 1

Už vieme, že v komplexných číslach má polynóm $x^n - 1$ práve n koreňov. Ľahko overíme, že sú všetky navzájom rôzne, a dokonca ich vieme ľahko explicitne zapísať. V komplexnej rovine týchto n komplexných čísel leží na jednotkovej kružnici a tvoria vrcholy pravidelného n-uholníka.

Keď zoberieme „kanonický“ koreň $\omega_n = \cos(2\pi/n) + i\sin(2\pi/n) = e^{2\pi i/n}$, všetky korene môžeme zapísať v nasledovnom tvare: $\omega_n^0,\ \omega_n^1,\ \ldots,\ \omega_n^{n-1}$.

Keď bude n z kontextu jasné, budeme namiesto ω_n písať len ω.

Ešte uvedieme jedno zaujímavé pozorovanie, ktoré nám bude neskôr užitočné: pre každé $n > 1$ platí, že súčet všetkých n-tých odmocnín z jednotky je nula. Toto je vidieť napr. zo symetrie: keď si označíme $s = \omega^0 + \omega^1 + \cdots + \omega^{n-1}$, tak vidíme, že $\omega s = \omega^1 + \omega^2 + \cdots + \omega^n = s$, a keďže $\omega \neq 1$, musí byť $s = 0$.

(Vyššie uvedený dôkaz si predstavte graficky. Zoberme v komplexnej rovine n bodov predstavujúcich všetky n-té odmocniny. Ich súčet musí byť nulový, lebo sa nezmení, keď všetky body zrotujeme o $2\pi/n$ radiánov okolo nuly.)

Platí dokonca aj všeobecnejšia vlastnosť. Uvažujme nasledujúci súčet všeobecnejšej geometrickej postupnosti: $s = \omega^0 + \omega^a + \cdots + \omega^{a(n-1)}$. Analogickou úvahou ako vyššie vidíme, že $\omega^a s = s$. Máme teda len dva prípady: buď $\omega^a = 1$ alebo $\omega^a \neq 1$. V prvom prípade je $s = 1+1+\cdots+1 = n$, v druhom prípade je nutne $s = 0$.

2 Dve reprezentácie polynómov

Majme polynóm $A(x) = \sum_{i=0}^{n-1} a_i x^i$. Pre jednoduchosť budeme predpokladať, že n je mocninou dvoch – ak treba, doplníme ďalšie členy s koeficientom 0.

Jednou reprezentáciou tohto polynómu je vektor jeho koeficientov: postupnosť $(a_0, a_1, \ldots, a_{n-1})$.

Ako sme práve videli, polynómy vieme reprezentovať aj iným spôsobom: keď poznáme n, môžeme si pevne zvoliť ľubovoľných n rôznych hodnôt x_i a polynóm si reprezentovať jeho funkčnými hodnotami pre tieto vstupy.

Príklad: Kvadratickú funkciu $f(x) = x^2 - 3x + 2$ si môžeme reprezentovať ako vektor koeficientov $(2, -3, 1, 0)$. Taktiež si ale môžeme zvoliť, že polynómy vyhodnocujeme v bodoch 0, 1, 2, 3 a následne našu konkrétnu funkciu f môžeme reprezentovať ako vektor funkčných hodnôt: $(2, 0, 0, 2)$. Náš polynóm je týmito funkčnými hodnotami jednoznačne určený. Presnejšie, existuje práve jeden polynóm nanajvýš tretieho stupňa, ktorý nadobúda v nami zvolených štyroch bodoch tieto funkčné hodnoty.

2.1 Násobenie v novej reprezentácii

Načo je dobrá táto nová reprezentácia? Dobre sa v nej polynómy násobí. Presnejšie, ak máme dva polynómy, ktorých súčin je ešte stále stupňa menšieho ako naše pevne zvolené n, tak naša reprezentácia stačí na jednoznačné určenie výsledku.

Príklad: Pokračujúc vo vyššie uvedenom príklade, zoberme lineárnu funkciu $g(x) = 4x + 3$. Jej nová reprezentácia (t.j. funkčné hodnoty v bodoch 0 až 3) je $(3, 7, 11, 15)$. Súčin $f \cdot g$ teraz vypočítame jednoducho: po zložkách vynásobíme naše dva vektory. Súčin $f \cdot g$ má teda nasledovnú reprezentáciu pomocou funkčných hodnôt: $(2 \cdot 3,\ 0 \cdot 7,\ 0 \cdot 11,\ 2 \cdot 15) = (6, 0, 0, 30)$.

Toto nás vedie k nasledujúcej schéme algoritmu na násobenie polynómov:

```
1. nájdi dostatočne veľké n (väčšie ako súčet stupňov f a g).
2. vyhodnoť polynómy f, g v ľubovoľných (ale tých istých) n bodoch
3. získané hodnoty po dvojiciach vynásob
4. interpoláciou získaj výsledok
```

Krok 2 ani krok 4 zatiaľ nevieme robiť lepšie ako v kvadratickom čase. V nasledujúcom texte si ukážeme výrazne efektívnejšie riešenie.

3 Diskrétna Fourierova transformácia (DFT)

Pripomeňme si, čo chceme: vedieť efektívne prevádzať medzi našimi dvomi reprezentáciami polynómov.

Vo všeobecnosti je to ťažký problém. Ak chceme v nejakom bode vyhodnotiť polynóm, ktorý má stupeň rádovo n, potrebujeme na to $\Theta(n)$ krokov výpočtu. Na našu konverziu potrebujeme takýto polynóm vyhodnotiť až v n rôznych bodoch. Ak by sme to robili postupne, trvalo by nám to $\Theta(n^2)$, čiže by sme už nemali šancu dostať efektívny algoritmus na násobenie.

Predchádzajúci odsek môžeme čítať aj ako návod: ak chceme efektívnejší algoritmus, musíme nájsť spôsob, ako daný polynóm vyhodnotiť *naraz vo veľa bodoch*.

Kde je nejaká voľnosť, ktorá by nám to umožnila? To, čo si môžeme zvoliť, je množina bodov, v ktorých naše polynómy vyhodnotíme. A ukáže sa, že šikovnou voľbou týchto bodov skutočne vieme dosiahnuť lepšiu časovú zložitosť.

Ako naše body si zvolíme... chvíľka napätia... áno, práve n-té komplexné odmocniny z jednotky. Ukáže sa, že práve ich symetria povedie k želanému zefektívneniu algoritmov.

3.1 Formálna definícia DFT

Pripomeňme si, že predpokladáme, že n je mocninou dvoch. Formálne je DFT transformácia na n-prvkových vektoroch komplexných čísel.

Vstupný vektor si označme $A = (a_0, a_1, \ldots, a_{n-1})$. Tento vektor interpretujeme ako (potenciálne komplexné) koeficienty polynómu $A(x) = \sum a_i x^i$. Výstupom transformácie DFT bude vektor $(a'_0, a'_1, \ldots, a'_{n-1})$, pre ktorý platí $a'_i = A(\omega_n^i)$. Inými slovami, výstupom DFT je reprezentácia toho istého polynómu prostredníctvom jeho funkčných hodnôt v n-tých komplexných odmocninách z jednotky.

3.2 Algoritmus výpočtu DFT

Základným trikom bude, že (podobne ako napr. pri MergeSorte) vyriešime jeden problém veľkosti n pomocou rekurzívneho riešenia dvoch problémov veľkosti $n/2$.

Ale nepredbiehajme, začnime od základného prípadu. Pre $n = 1$ je vstupom vektor (a_0) predstavujúci konštantný polynóm $A(x) = a_0$. Jeho hodnota v bode 1 (teda v jedinej prvej odmocnine z 1) je, prekvapivo, a_0. Výstupom je teda ten istý vektor, ktorý sme dostali na vstupe: vektor (a_0).

Ako teraz bude vyzerať všeobecný prípad? Majme nejaký polynóm $A(x)$. Jeho členy rozdelíme na dve kôpky podľa toho, či obsahujú párnu alebo nepárnu mocninu premennej x.

Formálne, definujme dva nové polynómy:

$$A_0(x) = a_0 + a_2 x + a_4 x^2 + \cdots + a_{n-2} x^{(n-2)/2}$$
$$A_1(x) = a_1 + a_3 x + a_5 x^2 + \cdots + a_{n-1} x^{(n-2)/2}$$

Každý z týchto polynómov je určený polovicou koeficientov pôvodného polynómu A. Pôvodný polynóm A teraz môžeme zapísať nasledovne:

$$A(x) = A_0(x^2) + x A_1(x^2)$$

Dostávame teda nasledujúci záver: na to, aby sme A vyhodnotili v nejakom bode x, nám stačí každý z polynómov A_0 a A_1 vyhodnotiť v bode x^2.

Toto nám vo všeobecnosti nepomôže. Ak by napríklad bolo naším cieľom vyhodnotiť A v bodoch $\{0, 1, 2, 3, \ldots, n-1\}$, dostali by sme z tejto úvahy, že každý z polynómov A_0 a A_1 potrebujeme vyhodnotiť v bodoch $\{0, 1, 4, 9, \ldots, (n-1)^2\}$. Tým sme si vôbec nepomohli. Máme síce dva menšie podproblémy, ale už vôbec nejde o inštancie toho istého problému. Totiž A_0 má len $n/2$ členov,

ale stále ho potrebujeme vyhodnotiť v n rôznych bodoch. Rekurzívne riešenie by v tomto prípade malo naďalej časovú zložitosť kvadratickú.

Tu sa ale ukáže vtip nášho hlavného triku – teda dôvod, prečo sme zvolili práve n-té komplexné odmocniny z jednej.

Pozrime sa napríklad, čo sa stane pre $n = 8$. Náš pôvodný polynóm A sme chceli vyhodnotiť v bodoch $\omega_8^0, \omega_8^1, \omega_8^2, \omega_8^3, \omega_8^4, \omega_8^5, \omega_8^6, \omega_8^7$. Polynómy A_0 a A_1 teda potrebujeme vyhodnotiť v nasledovných bodoch: $\omega_8^0, \omega_8^2, \omega_8^4, \omega_8^6, \omega_8^8, \omega_8^{10}, \omega_8^{12}, \omega_8^{14}$. No lenže ω_8^8 je z definície 1, a teda sa nám nejaké body opakujú. Presnejšie, druhá polovica bodov je úplne totožná s prvou: $\omega_8^8, \omega_8^{10}, \omega_8^{12}, \omega_8^{14}$ je to isté ako $\omega_8^0, \omega_8^2, \omega_8^4, \omega_8^6$.

Každý z polynómov A_0 a A_1 teda potrebujeme vyhodnotiť len v $n/2$ rôznych bodoch – tu ušetríme oproti predchádzajúcemu príkladu!

A v ktorýchže to $n/2$ bodoch ich potrebujeme vyhodnotiť? Nuž, naše n je párne a zjavne platí $\omega_n^2 = \omega_{n/2}$. Inými slovami, tých nových $n/2$ bodov tvoria práve všetky $(n/2)$-te odmocniny z 1. (V našom príklade vidíme, že štvorcom ω_8 je zjavne ω_4. A teda hodnoty $\omega_8^0, \omega_8^2, \omega_8^4, \omega_8^6$ sú vlastne hodnoty $\omega_4^0, \omega_4^1, \omega_4^2, \omega_4^3$.)

Ešte inými slovami, pre polynómy A_0 a A_1 potrebujeme vyriešiť presne ten istý problém ako pre pôvodný polynóm A: chceme nájsť ich DFT.

Celú implementáciu DFT si teda môžeme zhrnúť nasledovne:

```
1. rozdeľ polynóm A na polynómy A0 a A1
2. rekurzívne nájdi DFT polynómu A0
3. rekurzívne nájdi DFT polynómu A1
4. v čase O(n) pomocou vzťahu A(x) = A0(x^2) + x A1(x^2) vypočítaj DFT polynómu A
```

Pre časovú zložitosť nášho algoritmu dostávame takú istú rekurenciu ako u triedenia MergeSort. Celý výpočet teda prebehne v čase $\Theta(n \log n)$.

3.3 Ukážková implementácia DFT

Nasleduje kus C++ kódu implementujúci vyššie popísaný algoritmus.

```
typedef long double rnumber;
typedef complex<long double> cnumber;

vector<cnumber> DFT(const vector<cnumber> &A) {
    int N = A.size();
    if (N == 1) return A;

    vector<cnumber> B[2];
    for (int n=0; n<N; ++n) B[n&1].push_back( A[n] );

    for (int i=0; i<2; ++i) B[i] = DFT(B[i]);

    rnumber arg = 2 * M_PI / N;
    cnumber omega ( cos(arg), sin(arg) );

    cnumber x = 1;
    vector<cnumber> answer(N);
    for (int n=0; n<N/2; ++n) { answer[n]       = B[0][n] + x * B[1][n]; x *= omega; }
    for (int n=0; n<N/2; ++n) { answer[n+(N/2)] = B[0][n] + x * B[1][n]; x *= omega; }
    return answer;
}
```

V tejto chvíli sa oplatí upozorniť, že táto implementácia sa líši od implementácií používaných v praxi. Tie obsahujú ešte niekoľko ďalších myšlienkových krokov, ktoré síce nezlepšia asymptotickú časovú zložitosť, ale napriek tomu celkom výrazne (aj 10×) zrýchlia výpočet.

Štandardná sada takýchto vylepšení zahŕňa vlastnú implementáciu komplexných čísel (trieda `complex<>` má v g++ dosť pomalú implementáciu) a niekoľko trikov ktoré zmenšia potrebný počet aritmetických operácií a dovolia nám dokonca celý výpočet implementovať in-place, teda bez pomocných polí. Bližšie detaily týchto úprav vynecháme, záujemcov odkážeme napr. na `http://e-maxx.ru/algo/fft_multiply` (a na Google Translate).

Za zmienku stojí ešte jedno konkrétne vylepšenie. Ak si pamätáte, DFT sme si formálne definovali ako transformáciu na vektoroch komplexných čísel. Za dodatočného predpokladu že všetky

vstupy sú reálne vieme DFT ďalej zjednodušiť. Výsledkom takéhoto zjednodušenia je potom tzv. *diskrétna kosínusová transformácia* (DCT).

3.4 Checkpoint

Zastavme sa na chvíľu a uvedomme si, kam sme sa už dostali. Naším cieľom je efektívne násobenie polynómov. Pripomeňme si, ako ho chceme robiť:

```
1. nájdi dostatočne veľké n (väčšie ako súčet stupňov f a g).
2. vyhodnoť polynómy f, g v ľubovoľných (ale tých istých) n bodoch
3. získané hodnoty po dvojiciach vynásob
4. interpoláciou získaj výsledok
```

Fíha, veď už prvé tri zo štyroch krokov vieme vypočítať v celkovom čase $\Theta(n \log n)$. Ostáva nám teda už len nejak vymyslieť posledný krok: interpoláciu.

3.5 Inverzná DFT

Lenže, čo je to vlastne interpolácia? To je veľmi jednoduché: ide o inverzné zobrazenie k zobrazeniu DFT. Pomocou DFT sme vedeli previesť koeficienty polynómu na jeho funkčné hodnoty, inverzné zobrazenie teda zoberie funkčné hodnoty a prevedie ich naspäť na koeficienty.

Pri hľadaní efektívneho spôsobu výpočtu inverznej DFT po druhýkrát prídu k slovu komplexné čísla. Ukáže sa, že vďaka našej šikovnej voľbe bude výpočet inverznej DFT *takmer identický s výpočtom samotnej DFT*.

V tejto chvíli je OK chvíľu zhlboka dýchať a upokojiť sa. Predchádzajúce tvrdenie je skutočne šokujúce a prudko netriviálne.

3.6 Inverzné lineárne zobrazenie

Pri hľadaní inverzného zobrazenia nám pomôže, keď si uvedomíme, že DFT nie je len tak hocijaké zobrazenie. Hoci sa to na prvý pohľad možno nezdá, ide o zobrazenie lineárne: vieme ho zapísať pomocou násobenia vstupného vektoru vhodne zvolenou maticou. Celé to bude vyzerá nasledovne:

$$(a_0, a_1, \ldots, a_{n-1}) \cdot \begin{pmatrix} 1 & 1 & 1 & \cdots & 1 \\ 1 & \omega & \omega^2 & \cdots & \omega^{n-1} \\ 1 & \omega^2 & \omega^4 & \cdots & \omega^{2(n-1)} \\ \vdots & \vdots & \vdots & \vdots & \vdots \\ 1 & \omega^{n-1} & \omega^{2(n-1)} & \cdots & \omega^{(n-1)(n-1)} \end{pmatrix} = (a'_0, a'_1, \ldots, a'_{n-1}) \qquad (1)$$

Maticu vyskytujúcu sa vo vyššie uvedenom vzorci označíme M. Všimnite si, že pre jednoduchosť všade píšeme ω namiesto ω_n. V tomto trende budeme v tejto časti veselo pokračovať aj ďalej, keďže všetky odmocniny, s ktorými budeme pracovať, budú n-té odmocniny. Všimnite si tiež, že všetky 1 v matici M sú vlastne ω^0.

Ak chceme nájsť inverzné zobrazenie, chceme vlastne nájsť inverznú maticu M^{-1}. Na to použijeme tzv. Delfskú metódu: výsledok uhádneme a následne si ho dokážeme :)

Dobrým kandidátom na inverznú maticu je matica nasledovná:

$$\overline{M} = \begin{pmatrix} 1 & 1 & 1 & \cdots & 1 \\ 1 & \omega^{-1} & \omega^{-2} & \cdots & \omega^{-(n-1)} \\ 1 & \omega^{-2} & \omega^{-4} & \cdots & \omega^{-2(n-1)} \\ \vdots & \vdots & \vdots & \vdots & \vdots \\ 1 & \omega^{-(n-1)} & \omega^{-2(n-1)} & \cdots & \omega^{-(n-1)(n-1)} \end{pmatrix}$$

Čo dostaneme, keď vynásobíme M a $\overline{M}$? Na políčku (i,j) bude skalárny súčin vektorov $(1,\omega^i,\omega^{2i},\ldots,\omega^{(n-1)i})$ a $(1,\omega^{-j},\omega^{-2j},\ldots,\omega^{-(n-1)j})$. Inými slovami, na políčku (i,j) dostaneme súčet geometrickej postupnosti $1+\omega^k+\omega^{2k}+\cdots+\omega^{(n-1)k}$, kde $k=i-j$.

No a tieto súčty už predsa poznáme (viď koniec časti 1.3). Pre $k=0$ (teda $i=j$, teda na hlavnej diagonále) je každý sčítanec rovný 1, a teda dostávame súčet n. Pre $k\neq 0$ (teda všade mimo hlavnej diagonály) dostávame súčet 0.

Matica $M\cdot\overline{M}$ je teda n-násobkom identity. Inými slovami, maticu M^{-1} dostaneme, keď všetky prvky matice $\overline{M}$ vydelíme n.

3.7 Algoritmus inverznej DFT

Ako nám ale práve uskutočnené pozorovanie pomôže počítať inverznú DFT efektívne?

V časti 3.3 sme si ukázali program, ktorý počíta DFT. To znamená, že výstupom tohto programu je ten istý vektor, ktorý by sme dostali, keby sme jeho vstup vynásobili maticou M.

My teraz chceme program, ktorý „násobí“ maticou $\overline{M}/n$. Ako bude vyzerať? To je jednoduché: vyzerať bude tak isto, len namiesto ω v ňom použijeme ω^{-1} (čiže $1/\omega$), no a úplne na konci (len raz, po úplnom dobehnutí celej rekurzie!) všetky prvky výstupného poľa vydelíme n.

Najjednoduchšie je doplniť tieto časti priamo do už existujúcej implementácie DFT. V nasledujúcej ukážke kódu uvádzame len zmenené riadky.

```
vector<cnumber> DFT(const vector<cnumber> &A, bool inverse=false) {
    // ...
    for (int i=0; i<2; ++i) B[i] = DFT(B[i],inverse);
    // ...
    rnumber arg = (inverse?-1:1) * 2 * M_PI / N;
    // ...
}

vector<cnumber> inverse_DFT(const vector<cnumber> &A) {
    int N = A.size();
    vector<cnumber> answer = DFT(A,true);
    for (int n=0; n<N; ++n) answer[n] /= N;
    return answer;
}
```

4 Rýchle násobenie polynómov

Hurá, zvíťazili sme. Keď máme dva polynómy, ktorých súčet stupňov je n, vieme ich vynásobiť v čase $O(n\log n)$. Stačí nám postupne:

1. Zvýšiť n na najbližšiu mocninu dvoch.
2. Každý polynóm doplniť nulami aby mal n koeficientov.
3. Na každom polynóme zvlášť spraviť DFT, čím ho prevedieme na jeho funkčné hodnoty v n-tých odmocinách z 1.
4. Nové reprezentácie oboch polynómov po zložkách vynásobiť, čím dostaneme reprezentáciu ich súčinu.
5. Inverznou DFT zistiť koeficienty súčinu.

4.1 Numerické problémy

Postup, ktorý sme si vyššie opísali, dokonale funguje v ideálnom matematickom svete. Keď sa však pozrieme na náš program, vidíme, že má do ideálneho matematického sveta ďaleko. V čom je problém? V tom, že floating-point čísla majú len obmedzenú presnosť a pri výpočtoch s nimi nutne dochádza k zaokrúhľovacím chybám.

Občas si s nimi vieme poradiť. Ak napríklad násobíme polynómy s reálnymi koeficientami, tak nám malé zaokrúhľovacie chyby v podstate nevadia. A čo ak násobíme polynómy s celočíselnými koeficientami? Nuž, tam zase vieme využiť túto dodatočnú informáciu. Začneme tým, že spravíme celý výpočet pomocou (nepresných floating-point) komplexných čísel. Na konci dostaneme nie-úplne-presné hodnoty hľadaných koeficientov. No a v tejto chvíli využijeme, že vieme, že

všetky výsledné koeficienty majú byť celé čísla – a tak každý koeficient jednoducho zaokrúhlime na najbližšie celé číslo.

Samozrejme, vyššie popísaný postup nie je univerzálny všeliek. Máme k dispozícii len konečne veľa bitov, a tak časom (pre dostatočne veľké n a dostatočne veľké hodnoty koeficientov) už jednoducho zaokrúhľovacie chyby začnú byť priveľké. Akonáhle sa priblížia k rádu jednotiek, bude s naším násobením celočíselných polynómov ámen.

V tejto chvíli by sa mohlo zdať, že je vhodné zaoberať sa presnejšie otázkou, čo si vlastne ešte môžeme dovoliť a čo už nie – teda napríklad nejakým odhadom toho, aké veľké n a aké veľké celočíselné koeficienty môžeme mať ak chceme zaručiť že vypočítame bezchybný výsledok. My sa však takouto otázkou zaoberať nebudeme, keďže všetky otázky týkajúce sa presnosti výpočtov neskôr vyriešime úplne iným spôsobom.

Zabudnime teda na zaokrúhľovacie chyby, vráťme sa do ideálneho matematického sveta a pozrime sa radšej na to, čo ešte vieme z DFT vyžmýkať.

5 Cyklická konvolúcia

Zostaňme ešte pri postupe popísanom v predchádzajúcej časti:

- zoberieme dve polia A a B dĺžky n
- pomocou DFT ich transformujeme na nové polia A' a B'
- vyrobíme nové pole C' v ktorom $c'_i = a'_i \cdot b'_i$
- inverznou DFT z poľa C' vyrobíme výstupné pole C

Čo vieme povedať o obsahu poľa C?

Už sme videli, že keď do polí A a B vyplníme koeficienty nejakých dvoch polynómov a doplníme ich vhodne veľa nulami, tak v poli C dostaneme koeficienty ich súčinu. Platí teda napríklad:

$$\begin{aligned} c_0 &= a_0 b_0 \\ c_1 &= a_0 b_1 + a_1 b_0 \\ c_2 &= a_0 b_2 + a_1 b_1 + a_2 b_0 \\ &\dots \end{aligned}$$

Vo všeobecnosti, každé c_i je tzv. *konvolúciou* nejakého kúsku poľa A s rovnako dlhým kúskom poľa B. (Inými slovami, ide o skalárny súčin dotyčného kúsku poľa A a reverzu dotyčného kúsku poľa B.)

Čo sa ale stane vo všeobecnom prípade, teda pre ľubovoľné dve vstupné polia A a B? Ukáže sa, že nám do vyššie uvedených vzťahov len pribudnú ďalšie členy, ktoré tam teraz nevidíme, lebo sú nulové – keďže aspoň jeden činiteľ v každom z nich je jedna z núl ktoré sme pri násobení polynómov dopĺňali na koniec polí A a B.

Totiž to, čo náš vyššie uvedený postup počíta, je v skutočnosti *cyklická konvolúcia* vstupných polí A a B. Formálne tým myslíme nasledovnú skutočnosť:

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

(Index $i - j$ uvažujeme modulo n. Teda napr. b_{-1} je b_{n-1}.)

Inými slovami, pre každé i platí, že hodnota c_i je skalárnym súčinom vektoru A a vektoru, ktorý dostaneme z B tak, že ho (cyklicky) prečítame v opačnom smere, začínajúc od indexu i.

Vyššie uvedené tvrdenie vieme pomerne ľahko dokázať priamym búšením do vzorcov. Je ale lepšie nahliadnuť jeho význam cez naše úvahy o interpolácii.

Označme $A(x)$ a $B(x)$ polynómy, ktorých koeficientmi sú polia A a B. Označme $C(x)$ súčin týchto dvoch polynómov. Keďže tentoraz už nemáme predpoklad o tom, že A a B končia dostatočne veľa nulami, vo všeobecnosti sa môže stať, že C má stupeň ostro väčší ako $n-1$.

Všimnime si, čomu sa rovná funkčná hodnota takéhoto polynómu $C(x)$ pre vstup ω^i.

$$C(\omega^i) = c_0 + c_1\omega^i + \cdots + c_{n-1}\omega^{(n-1)i} + c_n\omega^{ni} + c_{n+1}\omega^{(n+1)i} + \cdots$$

Teraz si môžeme všimnúť, že $\omega^{ni} = (\omega^n)^i = 1^i = 1$. A teda:

$$C(\omega^i) = c_0 + c_1\omega^i + \cdots + c_{n-1}\omega^{(n-1)i} + c_n + c_{n+1}\omega^i + \cdots$$

Definujme teraz nový polynóm $\overline{C}(x)$ stupňa (nanajvýš) $n-1$ nasledovne: $\forall i : \overline{c}_i = c_i + c_{n+i}$. Zjavne pre všetky i platí $C(\omega^i) = \overline{C}(\omega^i)$. Keď teda v našom algoritme (uvedenom na začiatku tejto časti) zoberieme hodnoty v poli C' a inverznou DFT z nich vypočítame výstup, dostaneme práve koeficienty polynómu $\overline{C}$.

Z toho môžeme sformulovať nasledovný záver: ak vyplníme do polí A a B koeficienty polynómov ktoré chceme vynásobiť ale nedoplníme za ne dostatočne veľa núl, koeficienty ich súčinu nám cyklicky pretečú: hodnota c_n sa pripočíta k hodnote c_0, hodnota c_{n+1} k hodnote c_1, a tak ďalej.

Teda vo výslednom poli C na indexe 0 nebude len sčítanec a_0b_0 (člen c_0) ale takisto všetky sčítance tvaru a_ib_{n-i} pre $1 \leq i < n$ (ktoré dokopy tvoria člen c_n). A tak ďalej.

6 Kombinatorika pomocou konvolúcie

Pomocou konvolúcie si teraz vyriešime netriviálnu algoritmickú úlohu.

6.1 Súčty dvoch prvkov

Majme pole $S = (s_0, s_1, \ldots, s_{n-1})$, pričom $n \leq 10^6$ a každé s_i je celé číslo z intervalu $[0, m)$. Koľko rôznych hodnôt nadobúdajú súčty $s_i + s_j$ (pričom i sa môže aj rovnať j)? Ktorú z nich nadobúdajú koľkokrát?

Priamočiare riešenie tejto úlohy by malo časovú zložitosť $\Theta(n^2)$. Je pomerne ťažké predstaviť si lepšie riešenie, no za dodatočného predpokladu že všetky s_i sú malé celé čísla sa nám to podarí spraviť s časovou zložitosťou $O(n + m \log m)$.

Začneme tým, že si do poľa A veľkosti m spočítame pre každú hodnotu počet jej výskytov v poli S. Na toto pole sa teraz budeme dívať ako na koeficienty polynómu $A(x)$. (Tento polynóm zvykneme nazývať *generujúcou funkciou* príslušnej postupnosti hodnôt.)

Čo sa stane keď polynóm $A(x)$ umocníme na druhú? Označme $B(x) = A^2(x)$. Čomu sa napríklad rovná koeficient b_3? Z definície násobenia polynómov je zjavné, že $b_3 = a_0a_3 + a_1a_2 + a_2a_1 + a_3a_0$. Vo všeobecnosti teda $b_k = \sum_i a_ia_{k-i}$.

Aký je kombinatorický význam hodnoty b_k? Ukážeme, že b_k je rovné počtu spôsobov, ktorými môžeme zapísať hodnotu k ako súčet dvoch prvkov poľa S. Začneme tým, že si vyberieme hodnotu i prvého z nich. Tým je jednoznačne určené, že druhý má hodnotu $k-i$. Hodnota i má v poli S presne a_i výskytov, hodnota $k-i$ ich má a_{k-i}. A keďže každý výskyt prvej môžeme skombinovať s každým výskytom druhej hodnoty, zodpovedá konkrétnej voľbe i práve a_ia_{k-i} možností. No a celkový počet možností dostaneme ako sumu cez všetky i.

Takto teda dostávame odpovede na naše otázky. Dosiahnuteľné súčty zodpovedajú práve všetkým nenulovým koeficientom polynómu B. Hodnoty dotyčných koeficientov predstavujú počty spôsobov ktorými sa dotyčné súčty dajú dosiahnuť.

6.2 Platenie mincami

Veľa kombinatorických úvah súvisiacich s opakovaním nejakého procesu vieme sformulovať pomocou generujúcich funkcií a ich konvolúcií. Ukážeme si ešte jeden príklad.

Uvažujme polynóm $E(x) = x^1 + x^2 + x^5 + x^{10} + x^{20} + x^{50} + x^{100} + x^{200}$. Exponenty v tomto polynóme zodpovedajú nominálnym hodnotám euro mincí (v centoch). Zoberme teraz nový polynóm

$(E(x))^{1000}$ a pozrime sa na koeficient pri x^{4700}. Aký je jeho kombinatorický význam? Tento koeficient je zjavne rovný počtu spôsobov, ktorými môžeme do automatu postupne nahádzať presne 1000 mincí tak, aby ich celková hodnota bola 47 eur.

7 Číselno-teoretická transformácia NTT

Ako už vieme, DFT môže mať problémy s presnosťou výpočtov. Navyše nie je úplne triviálne odhadnúť, aké veľké tie problémy sú – kedy už nastanú a kedy ešte nie. V tejto časti si ukážeme, ako sa týchto starostí raz a navždy zbaviť: nahradíme DFT v podstate ekvivalentným algoritmom, ktorý ale bude všetko počítať v celých číslach. Presnejšie, vo vhodne zvolenej modulárnej aritmetike.

7.1 Konečné polia s prvočíselnou veľkosťou

Počítanie modulo prvočíslo je fajn. Vieme klasickým spôsobom sčítať, odčítať, aj násobiť. A vďaka existencii inverzných prvkov vieme aj deliť.

Inverzné prvky modulo prvočíslo je najlepšie počítať pomocou malej Fermatovej vety: keďže $a^{p-1} \equiv 1 \pmod{p}$, tak inverzným prvkom k a je nutne a^{p-2}. A to vieme spočítať v $O(\log p)$ efektívnym umocnením.

7.2 Odmocniny z jednej

V konečných poliach vieme nájsť prvky, ktoré sa správajú podobne ako n-té komplexné odmocniny z 1. Presnejšie, aj v konečných poliach vieme riešiť rovnicu $x^n = 1$.

Zvoľme si napríklad $n = 2^{25}$. Chceme nájsť také konečné pole prvočíselnej veľkosti, v ktorom bude existovať nejaká n-tá odmocnina z jednej. Ako na to? Z Lagrangeovej vety („rád prvku delí rád grupy“) vieme, že pre každé x najmenšie i také, že $x^i \equiv 1 \pmod{p}$, delí $p-1$. Ak teda chceme prvok, pre ktorý bude najmenšie i mať hodnotu 2^{25}, potrebujeme p tvaru $a \cdot 2^{25} + 1$.

Príkladom takéhoto p je $p = 125 \cdot 2^{25} + 1 = 4\,194\,304\,001$. Toto je fajn prvočíslo, lebo je tesne menšie ako 2^{32}, takže sa príjemne zmestí do unsigned intu.

Existuje pre toto p naozaj 2^{25}-tá odmocnina z jednej? Skutočne áno. Metódou pokus-omyl ľahko nájdeme, že požadovanú vlastnosť má napr. $\omega = 199$. Všetky čísla $\omega^0, \omega^1, \omega^2, \ldots, \omega^{2^{25}-1}$ (počítajúc modulo p) sú navzájom rôzne, a následne $\omega^{2^{25}} = 1$. (V skutočnosti pri overovaní, či konkrétna hodnota ω vyhovuje, stačí overiť, že $\omega^{2^{24}}$ nie je 1 a že $\omega^{2^{25}}$ už je 1. Vidíte prečo?)

Našli sme teda konkrétne čísla $p = 125 \cdot 2^{25} + 1$ a $\omega = 199$, pre ktoré platí, že pri počítaní modulo p je ω „kanonickou“ 2^{25}-tou odmocninou z 1.

Všimnite si ešte, že (opäť, analogicky ako u komplexných čísel) tým, že poznáme $\omega_{2^{25}}$, poznáme aj príslušné kanonické n-té odmocniny z 1 pre všetky $n = 2^{24}, 2^{23}, \ldots, 2^1, 2^0$. Každá z nich je totiž (pochopiteľne, počítajúc modulo p) štvorcom predchádzajúcej.

Príklad: Nech $i = 2^{25}$ je najmenší kladný exponent, pre ktorý je $\omega^i = 1$. Zoberme $x = \omega^8$. Aké je najmenšie j, pre ktoré je $x^j = 1$? Zjavne $x^j = \omega^{8j}$, a teda hľadané najmenšie j je $i/8 = 2^{22}$. Dostávame teda, že ω^8 je kanonickou 2^{22}-hou odmocninou z 1.

7.3 Transformácia NTT a jej inverzná transformácia

Pomocou vhodných konštánt p a ω (ako napr. tých z predchádzajúceho textu) vieme teraz implementovať v celých číslach (presnejšie, počítajúc modulo p) transformáciu veľmi podobnú DFT. Aj význam tejto transformácie bude analogický: vstupom bude vektor A dĺžky n (ktorá je mocninou dvoch) a výstupom bude rovnako dlhý vektor A'. Ak A interpretujeme ako koeficienty polynómu (celé čísla zo $\mathbb{Z}_p$), tak A' sú funkčné hodnoty tohto polynómu v nami zvolených n-tých odmocninách z 1 (opäť, celých číslach, počítajúc modulo p).

Na výpočet transformácie NTT použijeme aj s chlpami presne ten istý kód ako pri výpočte DFT (viď časť 3.3). Jediný rozdiel bude v tom, že ako ω_n budeme brať vhodné mocniny nami nájdenej konštanty ω. A samozrejme všetky násobenia budú násobeniami celých čísel modulo p.

Malo by byť očividné, že tento výpočet spraví to isté ako v prípade DFT: vypočíta lineárne zobrazenie popísané rovnicou 1, čiže vyhodnotí polynóm $A(x)$ vo všetkých mocninách ω_n.

No a úplne rovnako to bude fungovať aj s inverznou transformáciou.

Tu sa len oplatí si uvedomiť, že aj v našej modulárnej aritmetike platí, že súčet geometrickej postupnosti: $s = \omega^0 + \omega^a + \cdots + \omega^{a(n-1)}$ je buď n (ak $\omega^a = 1$) alebo 0 (ak $\omega^a \neq 1$).

Na výpočet inverznej NTT teda môžeme smelo použiť ten istý kód ako na výpočet inverznej DFT. Opäť, jediná zmena bude, že ω_n^{-1} tentokrát predstavuje inverzný prvok k ω_n v $\mathbb{Z}_p$.

Teda vlastne ešte jeden detail. V inverznej DFT sme na konci všetky hodnoty delili n. Čo sa stane v inverznej NTT? Pochopiteľne, vydelíme všetky hodnoty n. Lenže keďže sme v modulárnej aritmetike, nejde o klasické delenie. Namiesto toho všetky hodnoty vynásobíme n^{-1}. (Keďže $p > n$, táto inverzná hodnota určite existuje.)

7.4 Presnosť výpočtov

DFT bola vhodná ak sme počítali s polynómami, ktoré mali reálne (alebo dokonca komplexné) koeficienty a neprekážali nám zaokrúhľovacie chyby. Keď sme však chceli DFT používať na výpočty s celými číslami, nastával problém: aby sme mali istotu, že pri zaokrúhľovaní nespravíme chyby, potrebovali by sme vedieť, ako veľké nepresnosti môžu počas výpočtu nastať. No a výpočty s floating-point číslami, to je alchýmia do ktorej sa nechceme púšťať ak nemusíme.

NTT bude naopak veľmi vhodná práve (a len) na výpočty s polynómami ktoré majú celočíselné koeficienty. Tým, že všetky výpočty robíme modulo p, je všetko jasné: kým sú všetky výstupy z intervalu $[0, p)$, je všetko v poriadku, nikde nič nepretečie a výsledky vypočítame presne. (Teda napríklad ak len jednoducho násobíme dva polynómy s nezápornými celými koeficientami a vieme, že ich koeficienty sú také malé, že každý koeficient ich súčinu bude menší ako p, vieme, že výsledok dostaneme presne.)

Zamyslime sa teraz, čo môžeme robiť v dvoch prípadoch, kedy vyššie uvedené tvrdenie neplatí.

Prvým prípadom budú polynómy, ktoré síce majú celočíselné koeficienty, ale niektoré z nich môžu byť aj záporné.

Pre takéto polynómy všetko naďalej bez problémov funguje, všetky výpočty vieme naďalej robiť modulo p. Napr. hodnotu -47 reprezentujeme počas výpočtov ako $p - 47$. Na presnosť výsledku nám stačí, aby (pri výpočte v celých číslach bez modulovania) každá z výstupných hodnôt mala absolútnu hodnotu menšiu ako $p/2$. Potom je totiž zjavné, ktoré výstupné hodnoty majú byť kladné a ktoré záporné.

Druhým prípadom budú situácie, kedy niektoré výstupné hodnoty tesne ale predsa pretečú cez p. Čo vieme robiť v takomto prípade? Najjednoduchším riešením je zopakovať celý výpočet s inou hodnotou p a následne správne výsledky určiť pomocou čínskej zvyškovej vety.

Iné možné riešenie situácie kedy nám tesne nestačí presnosť je použiť obe verzie DFT: aj spojitú v doubloch (na určenie približného riešenia) aj diskrétnu modulo najväčšie p ktoré si vieme dovoliť (na následné určenie jedinej exaktnej hodnoty ktorá dáva správny zvyšok a sedí na približnú hodnotu zistenú v prvom kroku).

8 Ďalšie príklady použitia

V tejto poslednej časti si ešte ukážeme niekoľko ďalších príkladov použitia transformácií DFT a NTT. Vo všetkých prípadoch budeme používať nejakú variáciu násobenia polynómov.

8.1 Skalárne súčiny

Už sme si mohli viackrát všimnúť, že konvolúcia vlastne predstavuje skalárne súčiny jednej postupnosti s reverzom druhej. Čo ak potrebujeme počítať obyčajné skalárne súčiny? Pomoc je samozrejme jednoduchá: reverzneme jednu zo vstupných postupností.

Presnejšie, spravme nasledovnú zmenu: pred jeho spustením preusporiadame pole B do poradia $(b_0, b_{n-1}, b_{n-2}, \ldots, b_2, b_1)$. Inými slovami, B sme reverzli a zvolili sme si jeho konkrétnu cyklickú rotáciu. Čo dostaneme, keď teraz spustíme algoritmus z časti 5 ktorý počítal cyklickú konvolúciu A a B? Vo výstupe tentoraz bude platiť nasledovné:

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

Každé c_i je skalárnym súčinom pôvodného A s pôvodným B zrotovaným o i pozícií.

8.2 Najlepší match vzorky

Majme reťazce H a N (haystack, needle) nad abecedou $\{C, G, A, T\}$. Chceme nájsť taký offset v H, na ktorom začína podreťazec dĺžky $|N|$ ktorý sa najviac podobá na N – presnejšie, má od neho najmenšiu Hammingovu vzdialenosť. Inými slovami, chceme nájsť taký podreťazec H, ktorý sa s N na čo najviac miestach zhoduje.

Naše riešenie začneme tým, že sa sústredíme len na výskyty písmena C. Utvorme si z reťazca H pole A tak, že každé C zmeníme na 1 a každé iné písmeno na 0. Analogicky „preložme“ reťazec N na pole B.

Obe polia doplníme na konci nulami. Následne spravíme vyššie popísaný výpočet, ktorý nám vypočíta do poľa C ich všetky skalárne súčiny. Ľahko nahliadneme, že hodnota c_i nám hovorí: keby si N priložil ku H začínajúc od pozície i, bude existovať presne c_i pozícií, na ktorých je znak C naraz v oboch reťazcoch.

Vyššie popísaný postup zopakujeme pre každé iné písmeno abecedy a získané počty nasčítame. Takto pre každý offset zistíme celkový počet zhodných znakov medzi N a príslušným podreťazcom H.

8.3 Korelácie a autokorelácie

Najjednoduchšia forma koeficientu korelácie dvoch vektorov je jednoducho kosínus uhla ktorý zvierajú. Inými slovami, je to ich skalárny súčin vydelený ich veľkosťami.

DFT nám dáva výborný spôsob ako naraz vypočítať veľa skalárnych súčinov. Vďaka tomu vieme efektívne riešiť napr. otázky nasledovného typu: „pre danú dlhú postupnosť A a kratšiu postupnosť B nájdi v A úsek ktorý najviac koreluje s B“. Jediné, čo treba navyše vedieť robiť, je pre každý úsek $A[i : i + len(B)]$ povedať veľkosť príslušného vektoru. Toto vieme robiť v konštantnom čase po tom ako si na začiatku predpočítame prefixové súčty štvorcov prvkov poľa A.

Za samostatnú zmienku stojí autokorelácia. Čo sa stane, keď v predchádzajúcej úlohe ako B použijeme samotnú A, a budeme uvažovať aj tie offsety kedy sa prekrývajú len čiastočne? Rozmyslite si, že takto vypočítané koeficienty korelácie nesú užitočnú informáciu o periodicite dát v postupnosti A. (Čo to znamená, že pre nejaký offset vidíme veľkú koreláciu?)

8.4 Blur

Grafická operácia „blur“ je vlastne len obyčajným (len zväčša teda dvojrozmerným) skalárnym súčinom pôvodného obrázku s maticou váh. Ak chceme počítať blur s veľkým polomerom, vieme to efektívne spraviť pomocou DFT.