DFT a NTT

mišof

1 Interpolácia

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 menšieho ako n, ktorý cez všetky tieto body prechádza. Uvažujme nasledujúce polynómy:

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

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 hľadaného polynómu). Matica tejto sústavy je tzv. Vandermondova matica, o ktorej sa dá dokázať, že je vždy regulárna – a teda existuje práve jedno riešenie.

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 nad $\mathbb{C}$ má práve n koreňov. Každý nekonštantný polynóm nad $\mathbb{C}$ vieme zapísať v tvare súčinu koreňových činiteľov.

Ak by interpolačné polynómy boli dva, ich rozdielom by bol nekonštantný polynóm stupňa menšieho ako n, ale mal by aspoň n koreňov, čo je spor.

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: $(2, 0, 0, 2)$.

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. Dostávame teda, že súčin $f \cdot g$ má 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 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: $\omega_n^0,\ \omega_n^1,\ \ldots,\ \omega_n^{n-1}$.

TODO vlastnosti: súčet 0 a iné

4 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.

4.1 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 $(a_0, a_1, \ldots, a_{n-1})$. Tento vektor interpretujeme ako (potenciálne komplexné) koeficienty polynómu $A(x) = \sum a_i x^i$. Výstupom transformácie DFT bude vektor $(y_0, y_1, \ldots, y_{n-1})$, pre ktorý platí $y_i = A(\omega_n^i)$. Inými slovami, výstupom DFT je reprezentácia toho istého polynómu prostredníctvom jeho funkčných hodnôt v n-tých komplexných odmocninách z 1.

4.2 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 $\{0, 1, \ldots, n-1\}$, dostali by sme z tejto úvahy, že každý z polynómov A_0 a A_1 potrebujeme vyhodnotiť v bodoch $\{0, 4, 9, \ldots, (n-1)^2\}$. 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: $\omega_8^8 = \omega_8^0$, $\omega_8^{10} = \omega_8^2$, a tak ďalej.

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 $\omega_n^2 = \omega_{n/2}$. (Napr. štvorcom ω_8 je zjavne ω_4.) Inými slovami, tých nových $n/2$ bodov tvoria práve všetky $(n/2)$-te odmocniny z 1. 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 lineárnom čase pomocou vzťahu A(x) = A0(x^2) + x A1(x^2)
   vypočítaj DFT polynómu A

4.3 Ukážková implementácia DFT

Nasleduje kus C++ kódu.

```
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;
}
```

Vyššie uvedená implementácia je priamym prepisom algoritmu, ktorý sme si popísali v predchádzajúcej časti. Podobne ako u MergeSortu si vieme dokázať, že jej časová zložitosť je $\Theta(n \log n)$.

V tejto chvíli sa oplatí upozorniť, že táto implementácia sa líši od implementácií používaných v praxi. Tie obsahujú ešte niekoľko ďalších myšlienkových krokov, ktoré ale už nezlepšia asymptotickú časovú zložitosť. Vďaka in-place implementácii a redukcii množstva počítaných údajov vieme vcelku výrazne (aj $10\times$) zlepšiť konštantný faktor v časovej zložitosti. Navyše trieda `complex<>` má v g++ dosť pomalú implementáciu a oplatí sa ju nahradiť vlastnou triedou ktorá vie len potrebné operácie.

Bližšie detaily týchto úprav vynecháme, záujemcov odkážeme napr. na `http://e-maxx.ru/algo/fft_multiply` (a na Google Translate).

4.4 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.

4.5 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.

4.6 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:

$$(a_0, a_1, \ldots, a_{n-1}) \cdot \begin{pmatrix} 1 & 1 & 1 & \cdots & 1 \\ 1 & \omega & \omega^2 & \cdots & \omega^{n-1} \\ 1 & \omega^2 & \omega^4 & \cdots & \omega^{2(n-1)} \\ \vdots & \vdots & \vdots & \vdots & \vdots \\ 1 & \omega^{n-1} & \omega^{2(n-1)} & \cdots & \omega^{(n-1)(n-1)} \end{pmatrix} = (y_0, y_1, \ldots, y_{n-1})$$

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 $(1, \omega^{-j}, \omega^{-2j}, \ldots, \omega^{-(n-1)j})$.

Ľahko overíme, že na hlavnej diagonále (teda pre $i = j$) dostaneme na každom políčku hodnotu n. A čo dostaneme na políčku mimo hlavnej diagonály? Na každom takomto políčku bude súčet $1 + \omega^k + \omega^{2k} + \cdots + \omega^{(n-1)k}$, kde $k = i - j$. No a pre všetky prípustné nenulové hodnoty k ľahko nahliadneme, že tento súčet má hodnotu 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.

4.7 Algoritmus inverznej DFT

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

V časti 4.3 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;
}
```

5 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.

5.1 Implementácia

TODO

5.2 Numerické problémy

TODO

6 Príklady použitia DFT

TODO

7 Číselno-teoretická transformácia NTT

Ako už vieme, DFT môže mať problémy s presnosťou výpočtov. Navyše nie je úplne triviálne odhadnúť, čo si môžeme dovoliť a čo už 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

TODO

7.2 Odmocniny z jednej

TODO

7.3 Transformácie NTT a inverzná NTT

TODO