Математичка гимназија

Матурски рад
из математике

Брза Фуријеова трансформација

Ученик
Стефан Шебез, 4д

Ментор
Вукашин Брковић

Београд, 28. мај 2026.

Садржај

1

Увод

У основној школи смо учили како да множимо полиноме. Објаснили су нам процедуру у којој сваки моном у првој загради множимо са сваким мономом у другој, а затим сабирамо коефицијенте уз исте степене. Уколико је степен првог полинома n, а другог m, јасно је да овај алгоритам ради у сложености $O(nm)$. Алгоритам брзе Фуријеове трансформације (FFT) нам решава проблем множења два полинома ефикасно, у сложености $O((n+m)\log(n+m))$. Џејмс Кули и Џон Туки су 1965. године објавили рад који описује овај алгоритам и тиме су га први широко популаризовали.

Интересантно је да је још 1805. године, више од 150 година пре њих, Карл Фридрих Гаус први открио алгоритам који је математички еквивалентан данашњем FFT покушавајући да израчуна орбиту астероида. Он сам није схватао значај свог открића и никада није објавио свој рад. Он је пронађен тек доста касније и објављен је постхумно 1866.

У основи алгоритма је математички појам дискретна Фуријеова трансформација (DFT), која је аналогна непрекидној Фуријеовој трансформацији. FFT ефикасно рачуна DFT и њен инверз. Неке од значајних примена FFT су у обради звука и сигнала, телекомуникацијама, у радарским и сонарним системима и другим областима. Стога многи сматрају да је ово најважнији алгоритам икада измишљен. Циљ овог рада је да прођемо кроз то како сам алгоритам функционише и да видимо неке његове примене у задацима из такмичарског програмирања.

2

Дискретна Фуријеова трансформација

2.1 Полиномна перспектива

Рецимо да имамо полином $P(x) = a_0 + a_1x + ... + a_{n-1}x^{n-1} \in \mathbb{C}[x]$. Њега можемо представити као низ коефицијената $(a_0, ..., a_{n-1})$. Међутим, постоји још један начин како то да урадимо. Ако одаберемо тачке $(x_0, y_0), ..., (x_{n-1}, y_{n-1})$ такве да је $P(x_i) = y_i$, тада из полиномне интерполације постоји јединствен полином P степена највише $n-1$ који задовољава дате услове.

Ово друго представљање значајно олакшава множење полинома. Ако имамо полиноме P, Q степена највише $n-1$ и бројеве $x_0, ..., x_{n-1}, y_0, ..., y_{n-1}, z_0, ..., z_{n-1} \in \mathbb{C}$ за које важи

$$P(x_i) = y_i, Q(x_i) = z_i$$

онда је

$$P(x_i)Q(x_i) = y_iz_i.$$

Приметимо да полином $P(x)Q(x)$ има степен највише $2n-2$. Због тога нам је за његову једнозначну реконструкцију потребно најмање $2n-1$ вредности. Дакле, приликом множења полинома није довољно користити само n тачака, већ је потребно додати нулте коефицијенте на крај тако да број тачака буде довољно велики.

Сада, ако би постојао начин да се вратимо у прво представљање полинома добили бисмо коефицијенте полинома $P(x)Q(x)$. Пажљив одабир специјалних x_i ће нам омогућити да ефикасно прелазимо из једног представљања у друго. Узимајући да x_i буду комплексни n-ти корени јединице долазимо до **Дискретне Фуријеове трансформације**.

Дефиниција 2.1. Нека је $\omega = e^{\frac{2\pi i}{n}}$. За низ n комплексних бројева $a_0, ..., a_{n-1} \in \mathbb{C}$, Дискретну Фуријеову трансформацију DFT_a зовемо низ $(b_0, ..., b_{n-1}) \in \mathbb{C}^n$ такав да је

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

Дакле за $P(x) = a_0 + a_1x + ... + a_{n-1}x^{n-1}$ имамо

$$\mathrm{DFT}_a = (P(1), P(\omega^{-1}), P(\omega^{-2}), ..., P(\omega^{-(n-1)})).$$

Приметимо како се у дефиницији појављује знак − у експоненту. Ово је суштински само договор. Да уместо њега стоји знак + остатак приче као и алгоритам би били исти. Зашто смо баш овако дефинисали биће јасно у следећем одељку где посматрамо DFT из перспективе линеарне алгебре. Вреди напоменути да у неким изворима DFT и јесте дефинисан са знаком +.

2.2 Перспектива линеарне алгебре

Посматрајмо векторе $v_k = (1, \omega^k, \omega^{2k}, ..., \omega^{(n-1)k}) \in \mathbb{C}^n$. За $i \neq j$ скаларни производ вектора је

$$\langle v_i, v_j \rangle = \sum_{k=0}^{n-1} \omega^{ik}\overline{\omega^{jk}} = \sum_{k=0}^{n-1} (\omega^{i-j})^k = \frac{\omega^{(i-j)n} - 1}{\omega^{i-j} - 1} = 0,$$

а за $i = j$ је

$$\langle v_i, v_i \rangle = \sum_{k=0}^{n-1} \omega^{ik}\overline{\omega^{ik}} = \sum_{k=0}^{n-1} (\omega^{i-i})^k = \sum_{k=0}^{n-1} 1 = n.$$

Дакле, ови вектори формирају ортогоналну базу. Пребацивање из једне у другу ортогоналну базу је значајно лакше него у општем случају. То је зато што можемо да искористимо формулу за ортогоналну пројекцију вектора. Пројекција вектора v на вектор u је

$$\text{proj}_u(v) = \frac{\langle v, u \rangle}{\langle u, u \rangle} \cdot u.$$

Координата c_i вектора a у бази $v_0, \ldots, v_{n-1}$ добија се пројекцијом a на v_i, дакле

$$c_i = \frac{\langle a, v_i \rangle}{\langle v_i, v_i \rangle} = \frac{1}{n} \sum_{k=0}^{n-1} a_k \overline{\omega^{ik}} = \frac{1}{n} \sum_{k=0}^{n-1} a_k \omega^{-ik}.$$

Видимо да важи $b_i = nc_i$ за свако $0 \leq i < n$. Дакле, DFT_a је представљање вектора a у бази $v_0, ..., v_{n-1}$, помножено са константом n. Овде се појављује негативан експонент због узимања комплексног конјугата при скаларном производу. Други начин да ово запишемо је

$$\begin{pmatrix} 1 & 1 & 1 & \ldots & 1 \\ 1 & \omega^{-1} & \omega^{-2} & \ldots & \omega^{-(n-1)} \\ 1 & \omega^{-2} & \omega^{-2\cdot 2} & \ldots & \omega^{-2(n-1)} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 1 & \omega^{-(n-1)} & \omega^{-(n-1)\cdot 2} & \ldots & \omega^{-(n-1)(n-1)} \end{pmatrix} \begin{pmatrix} a_0 \\ a_1 \\ a_2 \\ \vdots \\ a_{n-1} \end{pmatrix} = \begin{pmatrix} b_0 \\ b_1 \\ b_2 \\ \vdots \\ b_{n-1} \end{pmatrix}.$$

Ово је специјалан случај *Вандермондове* матрице.

Слично имамо да се из базе $v_0, \ldots, v_{n-1}$ враћамо у стандардну базу $e_0, ..., e_{n-1}$ формулом

$$a_i = \frac{\langle c, e_i \rangle}{\langle e_i, e_i \rangle} = \langle c, e_i \rangle = \langle \frac{1}{n} \sum_{k=0}^{n-1} b_k v_k, e_i \rangle = \frac{1}{n} \sum_{k=0}^{n-1} b_k \langle v_k, e_i \rangle = \frac{1}{n} \sum_{k=0}^{n-1} b_k \omega^{ik}.$$

Овиме долазимо до нама најважније теореме, о **инверзном** DFT.

Теорема 2.1. Нека је $b = \mathrm{DFT}_a$. Тада важи

$$a_i = \frac{1}{n} \sum_{k=0}^{n-1} b_k \omega^{ik},$$

и кажемо да је $a = \mathrm{IDFT}_b$ инверзни DFT од b.

Баш због тога што је могуће наћи IDFT и зато што је његова формула аналогна обичном DFT омогућава нам да направимо алгоритам који ефикасно прелази из једног у друго.

3

Брза Фуријеова трансформација

3.1 Алгоритам FFT

Применићемо честу технику у алгоритмима - завади па владај (divide and conquer). Нека нам је дат полином $P(x) = a_0 + a_1x + ... + a_{n-1}x^{n-1}$. Можемо додавати нулте коефицијенте на крај полинома док не добијемо да је n степен двојке. Дакле претпоставимо да n јесте степен двојке. Разматрајмо коефицијенте уз парне степене и коефицијенте уз непарне степене засебно. Тада можемо написати

$$P(x) = P_0(x^2) + xP_1(x^2),$$

где су

$$P_0(x) = a_0 + a_2x + \cdots + a_{n-2}x^{\frac{n}{2}-1},$$
$$P_1(x) = a_1 + a_3x + \cdots + a_{n-1}x^{\frac{n}{2}-1}.$$

Важи $\deg P_0 = \deg P_1 = \frac{n}{2} - 1$. Сада позивамо рекурзивно алгоритам да израчуна DFT_{P_0} и DFT_{P_1}. Наш циљ је да ова два низа комбинујемо у један. Нека је $\omega = e^{\frac{2\pi i}{n}}$. За $0 \leq i \leq \frac{n}{2} - 1$ имамо

$$\begin{aligned}
\mathrm{DFT}_P[i] = P(\omega^{-i}) &= P_0(\omega^{-2i}) + \omega^{-i}P_1(\omega^{-2i}) \\
&= P_0((\omega^2)^{-i}) + \omega^{-i}P_1((\omega^2)^{-i}) \\
&= \mathrm{DFT}_{P_0}[i] + \omega^{-i}\mathrm{DFT}_{P_1}[i], \\
\mathrm{DFT}_P[i + \frac{n}{2}] = P(\omega^{-(i+\frac{n}{2})}) &= P_0(\omega^{-2(i+\frac{n}{2})}) + \omega^{-(i+\frac{n}{2})}P_1(\omega^{-2(i+\frac{n}{2})}) \\
&= P_0(\omega^{-2i}) + \omega^{-i}\omega^{-\frac{n}{2}}P_1(\omega^{-2i}) \\
&= \mathrm{DFT}_{P_0}[i] - \omega^{-i}\mathrm{DFT}_{P_1}[i].
\end{aligned}$$

Једначине

$$\mathrm{DFT}_P[i] = \mathrm{DFT}_{P_0}[i] + \omega^{-i}\mathrm{DFT}_{P_1}[i],$$
$$\mathrm{DFT}_P[i + \frac{n}{2}] = \mathrm{DFT}_{P_0}[i] - \omega^{-i}\mathrm{DFT}_{P_1}[i],$$

се зову лептир (butterfly) трансформација. За $0 \leq i < n$ оне се могу краће написати и као

$$\mathrm{DFT}_P[i] = \mathrm{DFT}_{P_0}[i \mod \frac{n}{2}] + \omega^{-i}\mathrm{DFT}_{P_1}[i \mod \frac{n}{2}].$$

Ако је $f(n)$ комплексност алгоритма онда је $f(n) = 2f(\frac{n}{2}) + O(n)$ што јасно даје $f(n) = O(n \log n)$ по мастер теореми.

Овако смо израчунали DFT_P, а алгоритам можемо мало модификовати тако да рачуна и IDFT_P. Потребно је само уместо ω^{-i} ставити ω^i и на крају поделити са n због формуле за IDFT коју смо извели у прошлој глави.

3.2 Рекурзивна имплементација

Следећа је рекурзивна имплементација FFT.

```
#define ll long long
#define ld long double
#define cd complex<long double>
const ld PI=acos(-1);
void fft(vector<cd>&a,bool inv){
    int n=a.size();
    if(n<=1)return;
    vector<cd>A[2];
    for(int i=0;i<n;i++)A[i%2].push_back(a[i]);
    fft(A[0],inv);
    fft(A[1],inv);
    ld ang=-2*PI/n;if(inv)ang=-ang;
    cd w(1),wn(cos(ang),sin(ang));
    for(int i=0;i<n;i++,w*=wn){
        a[i]=A[0][i%(n/2)]+w*A[1][i%(n/2)];
        if(inv)a[i]/=2;
    }
}
```

Код 3.1: Рекурзивна имплементација FFT

Функција `fft` узима вектор комплексних бројева a и параметар `inv` који означава да ли је потребно израчунати DFT_a или IDFT_a, зависно да ли је `inv=false` или `inv=true` респективно. Одговор се чува у вектору `a`. Прво се рачуна величина вектора n, за коју гарантујемо да је степен двојке. Ако је $n \leq 1$ завршавамо рекурзију. У супротном рачунамо A_0, A_1, који представљају наше P_0, P_1, и позивамо рекурзију за њих. Онда дефинишемо `wn` $= e^{\pm \frac{2\pi i}{n}}$ и `w` $= 1$. Потом у петљи рачунамо `a[i]` по формули коју смо извели, ту је `w` $=$ `wn`^{i}. Уколико је `inv=true` још и поделимо `a[i]` са 2 да би се на сваком нивоу рекурзије по једном поделило са 2, што је укупно исто као да смо једном на крају поделили са n.

Множење два полинома се онда имплементира овако:

```
#define ll long long
#define ld long double
#define cd complex<long double>
vector<ll>mul(vector<ll>a,vector<ll>b){
    int n=1;
    while(n<(int)a.size()+(int)b.size()-1)n<<=1;
    vector<cd>A(n),B(n);
    for(int i=0;i<a.size();i++)A[i]=a[i];
    for(int i=0;i<b.size();i++)B[i]=b[i];
    fft(A,false);fft(B,false);
    for(int i=0;i<n;i++)A[i]*=B[i];
    fft(A,true);
    vector<ll>res(n);
    for(int i=0;i<n;i++)res[i]=(ll)round(A[i].real());
```

```
15      while(!res.empty()&&res.back()==0)res.pop_back();
16      return res;
17  }
```

Код 3.2: Множење полинома

Функција `mul` множи два полинома са `long long` коефицијентима, представљена векторима `a` и `b`. Прво рачунамо `n` који је први степен двојке већи или једнак броју коефицијената производа ова два полинома. То је неопходно да бисмо правилно израчунали све коефицијенте. `A` је копија вектора `a` са додатим нулама на крај тако да му је величина `n`. Исто важи и за `B`. Пошто је производ полином са целобројним коефицијентима, узимамо реалан део и заокружујемо га. Овде може доћи до грешке због прецизности. На крају бришемо нуле са краја које су сувишне.

Ова имплементација је кратка и једноставна, међутим постоји и итеративни приступ који је нешто компликованији али и знатно бржи.

3.3 Итеративни FFT

У итеративном приступу имамо само један вектор a и сва рачунања ћемо да извршавамо на њему, без значајне додатне меморије. За ово ће нам бити потребно да пермутујемо његове елементе на неки начин. Приметимо да на првом нивоу рекурзије сви бројеви на парним индексима иду у `A[0]`, а на непарним у `A[1]`. У следећем нивоу рекурзије је најмањи бит фиксиран (`0` или `1`), а овде сви бројеви који имају други најмањи бит `0` иду у `A[0]` и сви остали чији је други најмањи бит `1` иду у `A[1]`. Надаље се дешава иста ствар и закључујемо да, пошто прво улазимо у рекурзију за `A[0]`, треба да сваки индекс заменимо са индексом чији је бинарни запис обрнут. Овај редослед се зове bit-reversal permutation. Ако је $rev(x)$ број са обрнутим бинарним записом онда једино што треба да урадимо је да заменимо `a[x]` и `a[rev(x)]`. На пример, за $n = 8$ добијамо следећи поредак

$$a = \Big\{ \Big[(a_0, a_4), (a_2, a_6) \Big], \Big[(a_1, a_5), (a_3, a_7) \Big] \Big\}$$

јер имамо

$$\begin{aligned}
rev(0) &= rev(000_2) = 000_2 = 0 \\
rev(1) &= rev(001_2) = 100_2 = 4 \\
rev(2) &= rev(010_2) = 010_2 = 2 \\
rev(3) &= rev(011_2) = 110_2 = 6 \\
rev(4) &= rev(100_2) = 001_2 = 1 \\
rev(5) &= rev(101_2) = 101_2 = 5 \\
rev(6) &= rev(110_2) = 011_2 = 3 \\
rev(7) &= rev(111_2) = 111_2 = 7.
\end{aligned}$$

У првом пролазу спајамо први број са другим (a_0 са a_4), трећи са четвртим (a_2 са a_6), и тако даље. У другом пролазу спајамо прва два са друга два (a_0, a_4 са a_2, a_6), трећа два са четврта два (a_1, a_5 са a_3, a_7) и тако даље. Укупно имамо $\log_2 n$ пролаза.

3.4 Итеративна имплементација

Имплементација изгледа овако

```
#define ld long double
#define cd complex<long double>
const ld PI=acos(-1);
void fft(vector<cd>&a,bool inv){
    int n=a.size(),lg=0;
    while((1<<lg)<n)lg++;
    auto revbits=[&](int x){
        int y=0;
        for(int i=0;i<lg;i++)if(x>>i&1)y^=1<<(lg-1-i);
        return y;
    };
    for(int i=0;i<n;i++){
        int j=revbits(i);
        if(i<j)swap(a[i],a[j]);
    }
    for(int m=2;m<=n;m<<=1){
        ld ang=-2*PI/m;if(inv)ang=-ang;
        cd wm(cos(ang),sin(ang));
        for(int i=0;i<n;i+=m){
            cd w(1);
            for(int j=0;j<m/2;j++){
                cd x=a[i+j],y=w*a[i+j+m/2];
                a[i+j]=x+y;
                a[i+j+m/2]=x-y;
                w*=wm;
            }
        }
    }
    if(inv)for(auto &i:a)i/=n;
}
```

Код 3.3: Итеративна имплементација FFT

Као и пре, функција `fft` узима вектор `a` и `inv`. Прво рачунамо n и $\log_2 n$. Функција `revbits(x)` рачуна $rev(x)$ у $O(\log n)$ операција. Затим пролазимо редом од 0 до $n-1$ и замењујемо `a[i]` и `a[rev(i)]`. Овде пазимо да не заменимо исти пар двапут, зато имамо услов `i<j`. Даље имамо $\log_2 n$ пролаза. У сваком рачунамо `m` и `wm` $= e^{\pm\frac{2\pi i}{m}}$. Онда за сваки блок m узастопних бројева спајамо прву и другу половину блока на основу лептир трансформације. На крају поделимо све са n уколико је `inv=true`.

Овде постоји још једна оптимизација која је значајна за брзину. Приметимо како n пута позивамо функцију $revbits$, ово је укупно $O(n \log n)$ операција. Могуће је поређати a у тачан поредак у $O(n)$ операција. Идеја је да симулирамо додавање 1 на неки број као у обичном бинарном запису, једино што овде то радимо у обрнутом поретку. Имплементација изгледа овако

```
#define ld long double
#define cd complex<long double>
void fft(vector<cd>&a,bool inv){
    int n=a.size();
    for(int i=1,j=0;i<n;i++){
        int k=n;do j^=k>>=1;while(~j&k);
        if(i<j)swap(a[i],a[j]);
    }
    for(int m=2;m<=n;m<<=1){
```

```
10          ld ang=-2*PI/m;if(inv)ang=-ang;
11          cd wm(cos(ang),sin(ang));
12          for(int i=0;i<n;i+=m){
13              cd w(1);
14              for(int j=0;j<m/2;j++){
15                  cd x=a[i+j],y=w*a[i+j+m/2];
16                  a[i+j]=x+y;
17                  a[i+j+m/2]=x-y;
18                  w*=wm;
19              }
20          }
21      }
22      if(inv)for(auto &i:a)i/=n;
23  }
```

Код 3.4: Боља итеративна имплементација FFT

Док пролазимо кроз петљу крећемо са `i=1` јер се `a[0]` свакако не мења. Овде нам `j` представља наш број са обрнутим битовима. Почињемо од `k=n`. У сваком пролазу кроз `do while` прво поделимо `k` са 2, а затим XOR-ујемо j са k. Ово ће у првом кораку обрнути највећи бит, у другом други највећи бит, и тако даље. Због редоследа операција у језику `C++` се ово може написати као у коду. Увек проверавамо да ли је тренутни бит који смо обрнули 1, ако јесте завршавамо. То `~j&k` проверава јер оператор `~` обрће све битове у `j`, а `k` садржи само тај бит који нам треба. Остатак кода је исти.

Највећи бит се промени n пута, други највећи $\frac{n}{2}$, трећи највећи $\frac{n}{4}$ и тако даље. Због $n + \frac{n}{2} + \frac{n}{4} + \frac{n}{8} + \cdots = 2n$ уради се $O(n)$ операција у овом делу.

Такође је могуће и једном израчунати цео rev низ пре позивања `fft`. Ово може бити корисно у неким ситуацијама.

3.5 Поређење перформанси FFT имплементација

Итеративна имплементација је много бржа од рекурзивне из неколико разлога. У рекурзивној имплементацији сваки позив функције има додатан трошак због позива функције и рада са стеком. У функцији се праве нови вектори `vector<cd>A[2]`. Тиме се стално врши динамичка алокација меморије на хипу и током извршавања се укупно алоцира $O(n \log n)$ додатне меморије. У итеративној имплементацији није потребно много додатне меморије јер се операције раде на почетном низу. Најзад, велику разлику прави кеш меморија. У итеративном приступу се рачунања извршавају на меморијским локацијама које су постављене редом, за разлику од рекурзивног где су, поготово због алокације динамичке меморије, меморијске локације разбацане и потребно је скакати између места у меморији. Тиме се у рекурзији много чешће дешавају промашаји кеша (cache miss), који значајно успоравају извршавање.

Извршио сам тест у коме поредим брзине рекурзивне и обе итеративне имплементације. Тест је урађен на лаптопу Lenovo Legion 5 са процесором AMD Ryzen 7 6800H with Radeon Graphics, 3.2 GHz и 16GB RAM. Код је покренут командном линијом

```
g++ -O2 -static -Wl,--stack,268435456 -o main main.cpp
```

на Windows 11 Home верзији 25H2. Поредио сам брзину множења два полинома дужине n, за различите вредности n, са насумичним коефицијентима апсолутне

вредности до 10^9. Време је мерено у милисекундама. Бржа итеративна

n	Рекурзивна	Итеративна 1	Итеративна 2
10000	54	19	16
30000	109	42	35
50000	231	88	75
100000	484	192	167
200000	1185	493	401
500000	2113	890	786

Табела 3.1: Поређење времена извршавања различитих FFT имплементација у милисекундама.

имплементација (Итеративна 2) ради око 2.3 пута брже од рекурзивне (Рекурзивна), а око 1.2 пута брже од спорије итеративне имплементације (Итеративна 1).

4

Трансформација заснована на теорији бројева

Претпоставимо да су нам дата два полинома, као и прост број p. Наш циљ је да их помножимо по модулу p. Најлакши начин како то да урадимо је да урадимо FFT и да онда сваки коефицијент редукујемо модуло p. Ово може довести до грешке због прецизности. Трансформација заснована на теорији бројева, позната као number theoretic transform (NTT), гарантовано без грешке може ово да уради у сложености $O(n \log n)$.

Анализирајмо која својства комплексних бројева смо користили у FFT алгоритму. Због употребе линеарне алгебре нама је било неопходно да је $\mathbb{C}$ **поље**. Грубо речено, поље је математичка структура која подржава операције сабирања, множења и дељења над неким скупом. У општем случају, линеарна алгебра се ради над пољем. Кључна чињеница је то да је $\mathbb{F}_p$ поље, где су $\mathbb{F}_p = \{0, 1, \ldots, p-1\}$ и $+, \cdot$ сабирање и множење по модулу p. Овде је кључно то што је p прост. Једино под тим условом се може дефинисати $\frac{1}{x} \mod p$ за све $x \not\equiv_p 0$. Наиме, по малој Фермаовој теореми је $a^{p-1} \equiv_p 1$ за све $a \not\equiv_p 0$. Онда је $\frac{1}{a} = a^{-1} \equiv_p a^{p-2}$.

Ослањали смо се увелико и на број $\omega = e^{\frac{2\pi i}{n}}$. Оно што нам је било битно за ω је да важи $\omega^n = 1$ и да $\omega^k \neq 1$ за $k < n$. Такав број се назива ω **примитивни** n-ти корен јединице. Кључно нам је било да су $1, \omega, \omega^2, \ldots, \omega^{n-1}$ сва различита решења једначине $x^n = 1$. Ова чињеница следи из тога да је ω примитивни n-ти корен јединице.

Дакле, нама треба постојање $\omega \in \mathbb{F}_p$ такво да је $\omega^n \equiv_p 1$ и $\omega^k \not\equiv_p 1$ за $k < n$. То значи да нам је потребан ω чији је **поредак** по модулу p једнак n. За неко $a \in \mathbb{F}_p$, $a \neq 0$, поредак броја a по модулу p је најмање $n \in \mathbb{N}$ такво да је $a^n \equiv_p 1$ и пишемо $ord_p(a) = n$. Важна чињеница око поретка је да је

$$a^m \equiv_p 1 \iff ord_p(a) \mid m.$$

Пошто је $a^{p-1} \equiv_p 1$ имамо $ord_p(a) \mid p-1$. Дакле, сваки могући поредак је делилац броја $p-1$. Доказано је и супротно, за сваки делилац $d \mid p-1$ постоји $a \in \mathbb{F}_p$ такав да је $ord_p(a) = d$. Посебно је битно постојање броја g таквог да је $ord_p(g) = p-1$, њега називамо **примитиван корен** или **генератор**.

Потребно нам је да важи $\omega^n \equiv_p 1$, то јест $n = 2^m \mid p-1$. Ово значи да нам треба прост број облика $p = c \cdot 2^k + 1$ за неко велико k. Стандардан избор у такмичарском програмирању је $p = 998\,244\,353 = 119 \cdot 2^{23} + 1$. Овај модул је одличан за практичне сврхе из неколико разлога. Прво, $k = 23$ нам омогућава да радимо

трансформације са дужином до $2^{23} \approx 8.3 \cdot 10^6$ што је углавном довољно велико. Поред тога модул је мањи од 10^9 што омогућава да множимо без проблема да бројеви изађу из оквира `long long`. Најзад, може се проверити да је 3 један његов примитиван корен. Пракса је да се, уколико је у задатку потребно израчунати нешто по модулу, узима баш овај модул, поготово ако решење користи FFT.

Нека је g примитивни корен по модулу p. За $n = 2^m$ тврдимо да $g^{\frac{p-1}{n}} = a$ има поредак n. Јасно је да је тај поредак највише n због $a^n \equiv_p g^{p-1} \equiv_p 1$. Претпоставимо супротно, нека је $0 < l = ord_p(a) < n$. Тада је $1 \equiv_p a^l \equiv_p g^{\frac{(p-1)l}{n}}$ што је немогуће јер је $\frac{(p-1)l}{n} < p - 1$. Дакле $g^{\frac{p-1}{n}}$ има поредак n, то јест он је n-ти примитиван корен јединице.

NTT ради аналогно као FFT. Једина разлика је у томе како рачунамо ω и то што све операције радимо по модулу. Потребно је имплементирати и бинарно степеновање. Имплементација је следећа.

```
#define ll long long
const int mod=998244353;
ll Pow(ll a,ll b){
    ll res=1;
    while(b){
        if(b&1)res=res*a%mod;
        a=a*a%mod;
        b>>=1;
    }
    return res;
}
void ntt(vector<ll>&a,bool inv){
    int n=a.size();
    for(int i=1,j=0;i<n;i++){
        int k=n;do j^=k>>=1;while(~j&k);
        if(i<j)swap(a[i],a[j]);
    }
    for(int m=2;m<=n;m<<=1){
        ll wm=Pow(3,(mod-1)/m);
        if(!inv)wm=Pow(wm,mod-2);
        for(int i=0;i<n;i+=m){
            ll w=1;
            for(int j=0;j<m/2;j++){
                ll x=a[i+j],y=w*a[i+j+m/2]%mod;
                a[i+j]=(x+y)%mod;
                a[i+j+m/2]=((x-y)%mod+mod)%mod;
                w=w*wm%mod;
            }
        }
    }
    if(inv){
        ll n1=Pow(n,mod-2);
        for(auto &i:a)i=i*n1%mod;
    }
}
vector<ll>mul(vector<ll>a,vector<ll>b){
    int n=1;
    while(n<(int)a.size()+(int)b.size()-1)n<<=1;
    a.resize(n);b.resize(n);
    ntt(a,false);ntt(b,false);
    for(int i=0;i<n;i++)a[i]=a[i]*b[i]%mod;
    ntt(a,true);
```

```
43      while(!a.empty()&&a.back()==0)a.pop_back();
44      return a;
45  }
```

Вреди напоменути да, због тога што смо дефинисали DFT са негативним експонентом, у коду узимамо модуларни инверз уколико је `inv==false` а не у обрнутом случају.

Неки од корисних простих модула за NTT су

$$p = 998\,244\,353 = 119 \cdot 2^{23} + 1, g = 3,$$
$$p = 167\,772\,161 = 5 \cdot 2^{25} + 1, g = 3,$$
$$p = 469\,762\,049 = 7 \cdot 2^{26} + 1, g = 3,$$
$$p = 7\,340\,033 = 7 \cdot 2^{20} + 1, g = 3,$$
$$p = 1\,004\,535\,809 = 479 \cdot 2^{21} + 1, g = 3.$$

Уколико нам је дат модул који није погодан за NTT, попут $p = 10^9 + 7$, постоји неколико опција шта можемо радити. Углавном је најбоље да радимо обичан FFT или да урадимо NTT са неколико добрих модула па да одговор нађемо користећи Кинеску теорему о остацима. Оба решења захтевају да коефицијенти полинома не постану превише велики. На пример, ако бисмо узели $p_1 = 998\,244\,353, p_2 = 1\,004\,535\,809, p_3 = 469\,762\,049$ могли бисмо да радимо са коефицијентима величине до $p_1 p_2 p_3 \approx 4.71 \cdot 10^{26}$. Ово би захтевало велику пажљивост. Морали бисмо да користимо `__int128` и да пазимо на негативне бројеве.

5

Конволуције

5.1 Дискретна и непрекидна конволуција

Анализирајмо операцију множења два полинома. Нека су нам дати полиноми $P(x) = a_0 + a_1x + \cdots + a_nx^n, Q(x) = b_0 + b_1x + \cdots + b_mx^m$. Тада је

$$P(x)Q(x) = \sum_{i=0}^{n}\sum_{j=0}^{m} a_ib_jx^{i+j} = \sum_{k=0}^{n+m} x^k \sum_{\substack{i+j=k \\ 0\leq i\leq n \\ 0\leq j\leq m}} a_ib_j.$$

Ако додефинишемо $a_i = 0$ за $i < 0$ и $i > n$, и аналогно за b, можемо написати

$$P(x)Q(x) = \sum_{k=-\infty}^{\infty} x^k \sum_{i+j=k} a_ib_j.$$

Тако добијамо дефиницију дискретне конволуције два низа.

Дефиниција 5.1. За два низа a, b кажемо да је $c = a * b$ дискретна конволуција њих уколико за свако k важи

$$c_k = \sum_{i+j=k} a_ib_j.$$

Дакле, можемо сматрати да смо користећи FFT рачунали конволуцију низова. То је још једна важна перспектива, поред полиномне и линеарне алгебре. Из теорије DFT-а о којој смо већ причали следи

$$\mathrm{DFT}(a * b) = \mathrm{DFT}(a) \cdot \mathrm{DFT}(b).$$

То је разлог зашто је DFT користан при раду са дискретном конволуцијом.

Поред множења полинома, конволуција се природно појављује и у вероватноћи. Ако су нам дате две дискретне случајне променљиве X и Y са неким расподелама вероватноћа, тада се расподела вероватноће $X+Y$ добија дискретном конволуцијом

$$P(X + Y = k) = \sum_{i+j=k} P(X = i)P(Y = j).$$

Потпуно аналогно је сабирање две непрекидне случајне променљиве, само се овог пута појављује непрекидна конволуција, која укључује интеграл две реалне

функције. За непрекидне случајне променљиве X, Y са густинама вероватноће f_X, f_Y имамо

$$f_{X+Y}(k) = \int_{-\infty}^{\infty} f_X(t) f_Y(k-t) dt.$$

Овде је, аналогно DFT, корисна обична Фуријеова трансформација. Аналогна једначина важи за две погодне функције $f, g : \mathbb{R} \to \mathbb{C}$

$$\mathcal{F}(f * g) = \mathcal{F}(f) \cdot \mathcal{F}(g),$$

где су

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

5.2 Још конволуција

Постоји мноштво примера где се појављују неке варијанте конволуција.

5.2.1 Дирихлеова конволуција

У теорији бројева постоји Дирихлеова конволуција две аритметичке функције $f, g : \mathbb{N} \to \mathbb{C}$

$$(f * g)(n) = \sum_{d|n} f(d) g(\frac{n}{d}) = \sum_{ij=n} f(i) g(j).$$

Она има велики значај, посебно у аналитичкој теорији бројева.

5.2.2 Конволуције са битовским операцијама

Можемо радити конволуцију и са битовским операцијама. За дате низове a, b дужине 2^n имамо

$$c_k = \sum_{i|j=k} a_i b_j,$$
$$c_k = \sum_{i\&j=k} a_i b_j,$$
$$c_k = \sum_{\substack{i|j=k \\ i\&j=0}} a_i b_j,$$
$$c_k = \sum_{i\oplus j=k} a_i b_j.$$

Све четири конволуције се могу израчунати брзо, у сложености $O(2^n n)$ или $O(2^n n^2)$.

Прва (OR) и друга (AND) конволуција се могу израчунати користећи зета и Мебијусове трансформације

$$z(f(x)) = \sum_{y \subseteq x} f(y),$$
$$\mu(f(x)) = \sum_{y \subseteq x} (-1)^{|x|-|y|} f(y).$$

Овде гледамо бројеве x, y као скупове, на основу свог бинарног записа. Може се проверити да су ове две трансформације инверзне користећи формулу укључења искључења. У такмичарском програмирању је техника брзог рачунања зета трансформације позната као sum over subsets (SOS) dp. Сложеност је $O(2^n n)$.

За трећу конволуцију (subset sum convolution) се користе исте две трансформације, једино је овде алгоритам знатно компликованији и сложености $O(2^n n^2)$.

За последњу (XOR) конволуцију је неопходан алгоритам брзе Волш-Адамарове трансформације ($FWHT$). Овај алгоритам има много више сличности са обичним FFT и компликованији је од приступа заснованог на зета трансформацији. Сложеност је $O(2^n n)$.

5.2.3 (min,+) и (max,+) конволуције

Од великог значаја су и $(\min, +)$ и $(\max, +)$ конволуције. Пошто су оне аналогне, причаћемо само о $(\min, +)$ варијанти. Приметимо како су нама за обичну дискретну конволуцију биле потребне операције $+$ и $\cdot$. Да бисмо ово генерализовали на произвољне операције потребно је да имамо нека основна својства наших нових $+$ и $\cdot$, попут асоцијативности, комутативности и дистрибутивности. Конкретно, желимо да је $(\mathbb{R}, +, \cdot)$ прстен. Може се проверити да је структура $(\mathbb{R} \cup \{\infty\}, \min, +)$ идемпотентни полупрстен. Дакле нема сва својства прстена, али их има довољно да бисмо могли смислено да дефинишемо конволуцију. Овде је операција min аналогна операцији $+$, а $+$ аналогна $\cdot$. Онда за две функције f, g имамо

$$(f * g)(k) = \min_{i+j=k} (f(i) + g(j)).$$

У општем случају није познат брз алгоритам за рачунање ове конволуције. То је због тога што је она врло различита од обичне конволуције. За примену FFT нам је требало да је $(\mathbb{C}, +, \cdot)$ поље, као и да постоји примитивно решење једначине $a^n = 1$. С обзиром на то да јасно ове две ствари не важе за $(\mathbb{R} \cup \{\infty\}, \min, +)$, није могуће урадити аналоган алгоритам FFT-у. Најбољи познати алгоритми су сложености $O(\frac{n^2}{\log n})$ и $O(\frac{n^2 (\log\log n)^3}{(\log n)^2})$ који не представљају велико побољшање од $O(n^2)$.

Међутим, у случају да су f и g конвексне функције, могуће је применити технику Минковски суме конвексних многоуглова. На овај начин се постиже сложеност $O(n)$, која је оптимална. У случају $(\max, +)$ конволуције потребно је да f и g буду конкавне. Такође је могуће постићи исту сложеност при слабијем услову, где је само једна функција конвексна или конкавна респективно. Пример задатка где се ова техника нетривијално примењује је Japanese Olympiad in Informatics Spring Camp (JOISC) 2018 day 4 problem 1 Candies.

Чињеница да је $(\mathbb{R} \cup \{\infty\}, \min, +)$ полупрстен се повремено може појавити и у другачијим задацима. На пример, било је задатака где је идеја применити бинарно степеновање матрице где је множење матрица дефинисано са min и $+$

операцијама уместо стандардног $+$ и $\cdot$. Ово је могуће због својства асоцијативности и дистрибутивности.

6

Примене

6.1 Задатак 1

Дати су $n \leq 10^5$ и низ $a_1, \ldots, a_n$ целих бројева где је $a_i \leq 10^5$. Треба исписати низ b који садржи све бројеве x такве да постоје $1 \leq i, j \leq n$ такви да је $x = a_i + a_j$.

Решење. Ово је директна примена FFT. Нека је $P(x) = x^{a_1} + x^{a_2} + \cdots + x^{a_n}$. Ако израчунамо $P(x)^2$ добићемо да је коефицијент уз x^k број начина да напишемо $k = a_i + a_j$. Дакле да бисмо израчунали b потребно је наћи све коефицијенте различите од нуле. Да бисмо избегли грешку у прецизности згодно је користити NTT. Међутим, коефицијенти полинома $P(x)^2$ представљају број парова (i, j) који дају одређену суму, па могу прећи модул 998 244 353. Ово решавамо тако што прво уклонимо дупликате из низа a. Тада су сви коефицијенти највише 10^5.

6.2 Задатак 2

(Codeforces 954I)
Претпоставимо да су нам дата два стринга s и t, и да су њихове дужине једнаке. Могуће је извршити следећу операцију произвољан број пута: можемо одабрати два различита карактера c_1 и c_2, и заменити свако појављивање карактера c_1 у оба стринга са c_2. Означимо растојање између стринга s и t као минималан број операција потребних да ови стрингови постану једнаки. На пример, ако је $s = abcd$ и $t = ddcb$, растојање између њих је 2 — можемо заменити свако појављивање карактера a са b, тако да s постане $bbcd$, а затим можемо заменити свако појављивање карактера b са d, тако да оба стринга постану $ddcd$. Дати су стрингови S и T сачињени од карактера a до f. За сваки подниз стринга S који се састоји од $|T|$ карактера потребно је одредити растојање између тог подниза и стринга T.

Решење. Рецимо да имамо два стринга s, t исте дужине и желимо да нађемо њихову дистанцу. Направимо граф где су карактери c_1, c_2 повезани ако постоји $0 \leq i < |s|$ тако да је $\{c_1, c_2\} = \{s_i, t_i\}$. Тада је за сваку повезану компоненту потребно урадити операцију на свим карактерима сем евентуално једног. Дакле одговор је број карактера минус број повезаних компоненти.

Нека су $n = |S|, m = |T|$. Фиксирајмо карактере c_1, c_2 и посматрајмо низове a, b где је $a_i = 1$ ако је $S_i = c_1$, а 0 у супротном. Слично је $b_i = 1$ ако је $T_{m-1-i} = c_2$, а 0

у супротном. Конволуцијом a и b добијамо $c = a * b$ за које важи (за $m-1 \leq i < n$) да је $c_i > 0$ уколико су c_1, c_2 повезани у графу од подниза $S[i-m+1, \ldots, i]$ и T. Дакле довољно је за сваки пар карактера израчунати по једну конволуцију и онда за сваки подниз са DFS наћи број компоненти. Ако је $K = 6$ број карактера решење је сложености $O(K^2 n \log n)$.

6.3 Задатак 3

(Codeforces 1096G)
Све аутобуске карте у Берланду имају своје бројеве. Број се састоји од n цифара (n је паран број). Само k децималних цифара $d_1, d_2, \ldots, d_k$ могу се користити за формирање бројева карата. Ако се цифра 0 налази међу овим цифрама, онда бројеви могу имати водеће нуле. На пример, ако је $n = 4$ и могу се користити само цифре 0 и 4, онда су 0000, 4004, 4440 исправни бројеви карата, док 0002, 00, 44443 нису. Карта је срећна ако је збир првих $n/2$ цифара једнак збиру преосталих $n/2$ цифара. Израчунајте број различитих срећних карата у Берланду. Пошто одговор може бити велики, испишите га по модулу 998 244 353.

Решење. Нека је $P(x) = x^{d_1} + \cdots + x^{d_k}$ и $Q(x) = P(x)^{\frac{n}{2}} = a_0 + a_1 x + \cdots + a_d x^d$. Тада је a_m број шифри дужине $\frac{n}{2}$ чији је збир цифара једнак m. Дакле одговор је $a_0^2 + \ldots a_d^2$. Да бисмо израчунали $Q(x)$ прво допуњавамо низ коефицијената P нулама док не добијемо исти број коефицијената као код Q. Затим ћемо израчунати DFT_P па, користећи да је он скуп вредности полинома, можемо само сваки број у DFT_P да степенујемо на $\frac{n}{2}$ да бисмо добили DFT_Q. Овде користимо бинарно степеновање. На крају само треба да израчунамо IDFT_Q. Сложеност је $O(n \log n)$.

6.4 Задатак 4

За дато $p = 998\,244\,353$ и $0 \leq n < p$ израчунати $n!$ по модулу p.

Решење. Очигледан приступ би био да само петљом пређемо кроз бројеве редом и помножимо их. Пошто n може бити велико, ово решење је превише споро. Идеја је да фиксирамо $m \approx \sqrt{p}$ и да дефинишемо $P(x) = (x+1)(x+2)\ldots(x+m)$. Применом технике завади па владај заједно са NTT можемо да израчунамо $P(x)$ у $O(m \log^2 m)$. То радимо тако што узмемо $m_1 \approx \frac{m}{2}$ и рекурзивно израчунамо $Q(x) = (x+1)\ldots(x+m_1)$ и $R(x) = (x+m_1+1)\ldots(x+m)$. Онда једном применом NTT израчунамо $Q(x) \cdot R(x) = P(x)$. Сада применом технике `multi-point evaluation` [1] можемо израчунати $P(0), P(m), P(2m), \ldots, P((\lfloor \frac{n}{m} \rfloor - 1) \cdot m)$ у сложености $O(m \log^2 m)$. Ако је $n = mq + r$, где је $0 \leq r < m$, имамо $n! = P(0)P(m)\ldots P((q-1)m) \cdot (mq+1)\ldots(mq+r)$. Ово рачунање је јасно $O(m)$, дакле свеукупно је алгоритам $O(m \log^2 m) = O(\sqrt{p} \log^2 p)$.

6.5 Задатак 5

(The 4th Universal Cup. Stage 1: Grand Prix of Korolyov, Problem B Domain Compression)

[1] Алгоритам који омогућава ефикасно израчунавање вредности полинома у више тачака истовремено, погледати [7].

Гоцо има стабло са $n \leq 10^5$ чворова. Научио је технику компресије домена, која функционише на следећи начин:

- Прво бира неки чвор v који претходно није обрисан из графа. Затим, за сваки пар чворова $u < w$ који су повезани са чвором v, он додаје неусмерену грану (u, w) уколико она претходно није постојала у графу. Након тога, брише чвор v из графа, заједно са свим гранама које су са њим инцидентне.

За свако k $(1 \leq k \leq n)$, њега занима укупан број грана у преосталим графовима за свих $\frac{n!}{(n-k)!}$ начина за извршавање k ових операција. Пошто одговори могу бити веома велики, исписати их по модулу 998 244 353.

Решење. Рецимо да смо фиксирали пар различитих чворова u, v и k и да желимо да видимо у колико случајева након примене k операција постоји грана $u-v$. Нека је $u-x_1-x_2-\cdots-x_{d-1}-v$ јединствени пут у стаблу између u и v, где је $d = dist(u, v)$. Приметимо да грана $u - v$ постоји ако и само ако је операција примењена на све $x_1, \ldots, x_{d-1}$ а није примењена на u ни v. Дакле број могућности је

$$\binom{k}{d-1} \cdot (d-1)! \cdot \binom{n-(d-1)-2}{k-(d-1)} \cdot (k-(d-1))! = \frac{k!}{(k-(d-1))!} \cdot \frac{(n-(d-1)-2)!}{(n-2-k)!} = \frac{k!}{(n-2-k)!} \cdot \frac{(n-d-1)!}{(k-d+1)!}.$$

Приметимо како ово не зависи од u и v већ само од њихове раздаљине d. Нека је $\text{paths}[d]$ број неуређених парова u, v са дистанцом d. Тада је одговор за k

$$\text{ans}[k] = \frac{k!}{(n-2-k)!} \cdot (\text{paths}[1]\frac{(n-2)!}{k!} + \text{paths}[2]\frac{(n-3)!}{(k-1)!} + \cdots + \text{paths}[k+1]\frac{(n-k-2)!}{0!}).$$

Ако дефинишемо

$$P(x) = (n-2)!\text{paths}[1] + (n-3)!\text{paths}[2] \cdot x + \cdots + 0!\text{paths}[n-1] \cdot x^{n-2},$$
$$Q(x) = \frac{1}{0!} + \frac{x}{1!} + \frac{x^2}{2!} + \cdots + \frac{x^{n-2}}{(n-2)!},$$

имамо да је коефицијент уз x^k од $P(x)Q(x)$ тачно $\text{paths}[1]\frac{(n-2)!}{k!} + \text{paths}[2]\frac{(n-3)!}{(k-1)!} + \cdots + \text{paths}[k+1]\frac{(n-k-2)!}{0!}$. Дакле

$$\text{ans}[k] = \frac{k!}{(n-2-k)!} \cdot [x^k](P(x)Q(x)), 1 \leq k \leq n-2.$$

Јасно је да ово можемо израчунати брзо користећи NTT. Специјални случајеви су $\text{ans}[n-1]$ и $\text{ans}[n]$ који су јасно оба 0. Остаје још израчунати вредности $\text{paths}[k]$.

За ово ћемо употребити технику центроидне декомпозиције. Одаберимо центроид c овог стабла. По дефиницији, то је чвор такав да ако га избришемо стабло се подели у неколико подстабала које је свако бар дупло мање од почетног стабла. Хоћемо да израчунамо колико има путева сваке дужине који пролазе кроз чвор c. За i-то подстабло направимо полином $F_i(x)$ где је

$[x^j]F_i(x)$ број чворова u унутар тог подстабла на дистанци j од c. Да бисмо избројали и путеве који се завршавају у c дефинишимо $F_0(x) = 1$. Због временске сложености је потребно да сортирамо полиноме по степену, претпоставимо $\deg F_0 \leq \deg F_1 \leq \cdots \leq \deg F_m$, где је m број подстабала. Пролазимо редом кроз петљу и на paths$[k]$ додајемо $[x^k]((F_0(x) + \cdots + F_{i-1}(x)) \cdot F_i(x))$ за све $1 \leq i \leq m$. Ово броји све путеве који почињу у подстаблу i и завршавају се у неком од подстабала $1, \ldots, i-1$ или у c. Јасно је да ова множења радимо користећи NTT. Морамо да сортирамо полиноме по степену јер множење два полинома степена a и b ради у $O(\max(a,b)\log(\max(a,b)))$. У супротном би могло да се деси да је $\deg F_1(x) \approx \frac{n}{2}$ и да је такође $m \approx \frac{n}{2}$ што би било $O(n^2 \log n)$ операција. Када сортирамо сложеност је $O(\deg F_1 \log \deg F_1) + \cdots + O(\deg F_m \log \deg F_m) = O(n \log n)$. Даље бришемо c и позивамо рекурзивно свако од m подстабала. Тиме је укупна сложеност $O(n \log^2 n)$, јер се подстабла смањују дупло сваки пут, то јест дубина рекурзије је $\log n$.

7

Закључак

У овом раду смо прошли кроз основне идеје Дискретне Фуријеове трансформације и алгоритма брзе Фуријеове трансформације. Разматрали смо DFT из више перспектива и видели како се FFT ефикасно имплементира у пракси. Такође смо прошли кроз варијацију алгоритма, NTT, и видели зашто је она често практичнија у такмичарским задацима од FFT. Дотакли смо се и важног математичког појма конволуције и неких њених варијација. На крају смо кроз конкретне задатке видели како се ове технике примењују у такмичарском програмирању.

Литература

[1] Codeforces, *FFT*. https://codeforces.com/blog/entry/111371

[2] Codeforces, *OR Convolution for Common People*. https://codeforces.com/blog/entry/115438

[3] Codeforces, *Fastest way to get factorial modulo a prime*. https://codeforces.com/blog/entry/63491

[4] Codeforces, *Tutorial on Zeta Transform, Mobius Transform and Subset Sum Convolution*. https://codeforces.com/blog/entry/72488

[5] Codeforces, *Knapsack, Subset Sum and the (max,+) Convolution*. https://codeforces.com/blog/entry/98663

[6] CP-Algorithms, *Fast Fourier Transform*. https://cp-algorithms.com/algebra/fft.html

[7] CP-Algorithms, *Operations on polynomials and series*. https://cp-algorithms.com/algebra/polynomial.html

[8] Michael T. Heideman, Don H. Johnson, C. Sidney Burrus, *Gauss and the History of the FFT*. https://www.cis.rit.edu/class/simg716/Gauss_History_FFT.pdf

[9] Wikipedia, *Dirichlet convolution*. https://en.wikipedia.org/wiki/Dirichlet_convolution

[10] Wikipedia, *Discrete Fourier transform*. https://en.wikipedia.org/wiki/Discrete_Fourier_transform

[11] Wikipedia, *Fourier transform*. https://en.wikipedia.org/wiki/Fourier_transform