Fast Fourier Transform and Polynomials
Moscow International Workshop ACM ICPC 2017

Oleksandr Kulkov

Contents

1 Fast multiplication **1**
Karatsuba method . . . 1
Polynomial multiplication . . . 2
Interpolation . . . 2
Discrete Fourier Transform . . . 2
Cooley-tukey method . . . 3
Inverse transform . . . 3
Interlude . . . 4

2 Applications and variations of transform **5**
Convolution and correlation . . . 5
Number Theoretic Transform . . . 5
Chirp Z-transform . . . 5
Simultaneous transform of real polynomials . . . 5
Multiplication with arbitrary modulo . . . 6
Multidimensional Fourier transform . . . 6
Walsh-Hadamard transform and other convolutions . . . 6
Newton method for functions of polynomials . . . 7
Divide and Conquer . . . 8
Division and interpolation . . . 8

3 Exercises **9**
Knapsack . . . 9
Power sum . . . 9
Generalized Cooley-Tukey method . . . 9
Arythmetic progressions . . . 9
Distance between points . . . 9
Pattern matching . . . 9
Linear recurrents* . . . 9
Polynomial power* . . . 9

1 Fast multiplication

Karatsuba method Consider such common operation as multiplication of two numbers. Doing it by definition yields $O(n^2)$ operations. It was assumed for a long time that there are no better algorithms. First one to refute this was Karatsuba despite the fact that it is assumed that Gauss already used Fourier transform in his works.

Karatsuba's approach is concise and simple. Assume we multiplying $A = a_0 + a_1 x$ and $B = b_0 + b_1 x$. Then:

$$A \cdot B = a_0 b_0 + (a_0 b_1 + a_1 b_0)x + a_1 b_1 x^2 =$$
$$= a_0 b_0 + [(a_0 + b_0)(a_1 + b_1) - a_0 b_0 - a_1 b_1]x + a_1 b_1 x^2$$

For simplicity assume that numbers are given in binary representation and have length n. Then if we take $x = 2^k, k \approx n/2$, it will allow us to reduce original problem to three problems which are two times lesser than original one: for computation of $a_0 b_0$, $a_1 b_1$ and $(a_0 + b_0)(a_1 + b_1)$. Needed operations in this case can be estimated as

$$T(n) = 3T\left(\frac{n}{2}\right) + O(n) = O\left(n^{\log_2 3}\right) \approx O(n\sqrt{n})$$

Polynomial multiplication To end up having algorithm with better complexity we should note that any number can be represented as polynomial $A(2) = a_0 + a_1 \cdot 2 + \cdots + a_n \cdot 2^n$. To multiply two numbers we have to multiply corresponding polynomials and normalize them afterwards.

```
const int base = 10;
vector<int> normalize(vector<int> c) {
    int carry = 0;
    for(auto &it: c) {
        it += carry;
        carry = it / base;
        it %= base;
    }
    while(carry) {
        c.push_back(carry % base);
        carry /= base;
    }
    return c;
}

vector<int> multiply(vector<int> a, vector<int> b) {
    return normalize(poly_multiply(a, b));
}
```

Direct formula for polynomial product is as follows:

$$\left(\sum_{i=0}^{n} a_i x^i\right) \cdot \left(\sum_{j=0}^{m} b_j x^j\right) = \sum_{k=0}^{n+m} x^k \sum_{i+j=k} a_i b_j$$

Its computation yields $O(n^2)$ operations, which doesn't satisfy us.

Interpolation Assume we have set of points $x_0, \ldots, x_n$. Polynomial of degree n can be uniquely restored from the values in this points. One of possible ways to do this is to use Lagrange's interpolating polynomial:

$$y(x) = \sum_{i=0}^{n} y_i \prod_{j \neq i} \frac{x - x_j}{x_i - x_j}$$

Note that if we have values of two polynomials in some set of points we can in $O(n)$ calculate values in same points of their product. Also degree of product of polynomials of degree n and m equals to $n + m$, thus it is enough for us to calculate values of polynomials in any $n + m$ points.

```
void align(vector<int> &a, vector<int> &b) {
    int n = a.size() + b.size() - 1;
    while(a.size() < n) {
        a.push_back(0);
    }
    while(b.size() < n) {
        b.push_back(0);
    }
}

vector<int> poly_multiply(vector<int> a, vector<int> b) {
    align(a, b);
    auto A = evaluate(a);
    auto B = evaluate(b);
    for(int i = 0; i < A.size(); i++) {
        A[i] *= B[i];
    }
    return interpolate(A);
}
```

Unfortunately, direct computation of values yields $O(n^2)$ operations and interpolation needs even more but we can improve this estimate by using some set of points x_i with special properties.

Discrete Fourier Transform Consider that in field we're working in there is element w such that

$$\begin{cases} w^k = 1 & k = n \\ w^k \neq 1 & k < n \end{cases}$$

We will call it primary n^{th} root of unity. Such element yields many useful properties. In particular, all w^i are different for i from 0 to $k - 1$, also $w^m = w^{m \bmod n}$. Thus powers of w form remainders group modulo n if we consider sum operation. Values of polynomial in such points is called discrete Fourier transform.

Most often this roots are taken from complex number field. Due to Euler's formula

$$e^{i\varphi} = \cos\varphi + i\sin\varphi$$

We can sum up that all of them are of form $w^k = e^{i\frac{2\pi}{n}k}$. Besides that during polynomial multiplication we can consider roots of unity in other fields. For example one can use roots of unity from the ring of remainders modulo some prime number which will be considered latter.

Cooley-tukey method Consider polynomial in the form $P(x) = A(x^2) + xB(x^2)$, where $A(x)$ consists of coefficients near even powers of x and $B(x)$ consists of coefficients near odd powers. Let $n = 2k$. then

$$w^{2t} = w^{2t \bmod 2k} = w^{2(t \bmod k)}$$

Besides that w^2 is k^{th} root of unity, thus

$$P(w^t) = A\left(w^{2(t \bmod k)}\right) + w^t B\left(w^{2(t \bmod k)}\right)$$

This equation allows to reduce discrete transform of size n to two discrete transforms of size $\frac{n}{2}$ using $O(n)$ additional operations. This follows that total complexity if we use this formula will be

$$T(n) = 2T\left(\frac{n}{2}\right) + O(n) = O(n\log n)$$

Note that in this formula it is essential that n is even so it should be even on all recursion layers which follows that n is the power of 2.

```
typedef complex<double> ftype;
const double pi = acos(-1);

template<typename T>
vector<ftype> fft(vector<T> p, ftype w) {
    int n = p.size();
    if(n == 1) {
        return vector<ftype>(1, p[0]);
    } else {
        vector<T> AB[2];
        for(int i = 0; i < n; i++) {
            AB[i % 2].push_back(p[i]);
        }
        auto A = fft(AB[0], w * w);
        auto B = fft(AB[1], w * w);
        vector<ftype> res(n);
        ftype wt = 1;
        int k = n / 2;
        for(int i = 0; i < n; i++) {
            res[i] = A[i % k] + wt * B[i % k];
            wt *= w;
        }
        return res;
    }
}

vector<ftype> evaluate(vector<int> p) {
    while(__builtin_popcount(p.size()) != 1) {
        p.push_back(0);
    } // p.size() has to be the power of 2
    return fft(p, polar(1., 2 * pi / p.size()));
}
```

Inverse transform After we calculated needed values and pairwise multiplied values of first polynomial by values of the second one, we should conduct the inverse transform. One can note that all actions which were done during the direct transform were invertible so we can simply conduct inverse operations before going into recursion.

But there is simpler way. During the computations we actually applied matrix to the vector:

$$\begin{pmatrix} w^0 & w^0 & w^0 & w^0 & \dots & w^0 \\ w^0 & w^1 & w^2 & w^3 & \dots & w^{-1} \\ w^0 & w^2 & w^4 & w^6 & \dots & w^{-2} \\ w^0 & w^3 & w^6 & w^9 & \dots & w^{-3} \\ \vdots & \vdots & \vdots & \vdots & \ddots & \vdots \\ w^0 & w^{-1} & w^{-2} & w^{-3} & \dots & w^1 \end{pmatrix} \begin{pmatrix} a_0 \\ a_1 \\ a_2 \\ a_3 \\ \vdots \\ a_{n-1} \end{pmatrix} = \begin{pmatrix} y_0 \\ y_1 \\ y_2 \\ y_3 \\ \vdots \\ y_{n-1} \end{pmatrix}$$

Consider sum $\sum_{k=0}^{n-1}(w^i w^j)^k = \sum_{k=0}^{n-1} w^{(i+j)k}$. Any number of kind w^i satisfies

$$w^n = 1 \implies 1 - w^n = (1-w)(1 + w + w^2 + \cdots + w^{n-1}) = 0$$

Therefore,

$$\sum_{i=0}^{n-1}(w^i)^k = \begin{cases} n, & i = 0 \\ 0, & i \neq 0 \end{cases}$$

Thus this sum equals n if $i + j = 0$ or it equals 0 otherwise. This follows that

$$\frac{1}{n}\begin{pmatrix} w^0 & w^0 & w^0 & w^0 & \dots & w^0 \\ w^0 & w^{-1} & w^{-2} & w^{-3} & \dots & w^1 \\ w^0 & w^{-2} & w^{-4} & w^{-6} & \dots & w^2 \\ w^0 & w^{-3} & w^{-6} & w^{-9} & \dots & w^3 \\ \vdots & \vdots & \vdots & \vdots & \ddots & \vdots \\ w^0 & w^1 & w^2 & w^3 & \dots & w^{-1} \end{pmatrix}$$

This is inverse matrix which we will multiply vector by during the inverse transform. So during the inverse transform we have to calculate Fourier transform using w^{-1} instead of w and then divide everything by n.

```
vector<int> interpolate(vector<ftype> p) {
    int n = p.size();
    auto inv = fft(p, polar(1., -2 * pi / n));
    vector<int> res(n);
    for(int i = 0; i < n; i++) {
        res[i] = round(real(inv[i]) / n);
    }
    return res;
}
```

All the way we passed through is enough to multiply two integer numbers in $O(n \log n)$.

Interlude The code given above being correct and having $O(n \log n)$ complexity isn't actually ready to be used in contests. It has large constant and less numerically stable than effective transform implementations. It is so because this code perfectly illustrates notion of algorithm without any distractions and complications to optimize it.

Before descending to the next part the reader is dared to think of ways to improve accuracy and time expenses on their own. Most important things here are that inside the transform there should be no memory allocations, also it's recommended to work with pointers rather than vectors and roots of unity should be calculated beforehand. Also one should get rid of taking values by modulo. And furthermore one can note that instead of making transform with w^{-1} one can simply make it with w, and reverse elements from second to the last one. Here you can see one of relatively good implementations.

2 Applications and variations of transform

Convolution and correlation Assume we have $\{a_i\}_{i=0}^{n}$ and $\{b_j\}_{j=0}^{m}$. Then convolution is $\{c_k\}_{k=0}^{m+n}$ such that

$$c_k = \sum_{i=0}^{n} a_i b_{k-i}$$

As you see, it's just k^{th} coefficient of product. Correlation meanwhile is $\{d_k\}_{-n}^{m}$ such that

$$d_k = \sum_{i=0}^{n} a_i b_{k+i}$$

In both cases we assume that elements with out-of-bound indices equal zero. We can interpret correlation in two ways. On the one hand it is coefficients in product $A(x) \cdot B(x^{-1})$, which is shifted convolution of first and reversed second sequences. on the other hand, d_k is exactly scalar product of a_i and segment of b_j which starts in position k. Convolution and correlation are those things which you need to calculate most often in problems on Fourier transform.

Number Theoretic Transform As it was said earlier one can use roots of unity not only from complex numbers but also in some field. In this case we are interested in fields of remainder modulo some prime number. It is known that in each such field there is generating element g such that set of its powers equals set of field elements except zero.

Thus for each prime p there is $(p-1)^{th}$ root of unity. If also $(p-1) = c \cdot 2^k$ then g^c is $(2^k)^{th}$ root of unity in the field which allows us to use Cooley-Tukey method. Thus we are interested in numbers $p = c \cdot 2^k + 1$. Practice shows that there are actually pretty much such numbers for any k.

Chirp Z-transform Assume we have some number z and we want to calculate values of polynomial in points of kind $\{z^i\}_{i=0}^{n-1}$ that is we have to calculate $y_k = \sum\limits_{i=0}^{n-1} a_i z^{ik}$. To do this we should substitute $ik = \dfrac{i^2 + k^2 - (i-k)^2}{2}$ and then we'll have to calculate

$$y_k = z^{\frac{k^2}{2}} \sum_{i=0}^{n-1} \left(a_i z^{\frac{i^2}{2}} \right) z^{-\frac{(i-k)^2}{2}}$$

Which is up to $z^{\frac{k^2}{2}}$ multiplier convolution of two sequences

$$u_i = a_i z^{\frac{i^2}{2}},\ v_i = z^{-\frac{i^2}{2}}$$

Which can be calculated as the product of polynomials with such coefficients. But you should note that v_i defined also for negative i. This approach allows to calculate Fourier transform of arbitrary length in $O(n \log n)$.

Simultaneous transform of real polynomials Assume you have

$$A(x) = \sum_{i=0}^{n-1} a_i x^i,\ B(x) = \sum_{i=0}^{n-1} b_i x^i$$

With real coefficients. Consider $P(x) = A(x) + iB(x)$ and conjugated polynomial to it.

$$\overline{P(w^k)} = A(\overline{w^k}) - iB(\overline{w^k}) = A(w^{n-k}) - iB(w^{n-k})$$

It follows that for transforms of $A(x)$ and $B(x)$ takes place

$$\begin{cases} A(w^k) = \dfrac{P(w^k) + P(w^{n-k})}{2}, \\ B(w^k) = \dfrac{P(w^k) - P(w^{n-k})}{2i} \end{cases}$$

Simultaneous transform can be also done in reverse direction considering sequence $P(w^k) = A(w^k) + iB(w^k)$. After inverse transform of this sequence we will have polynomial $P(x) = A(x) + iB(x)$.

Multiplication with arbitrary modulo We have to multiply two polynomials and then output result coefficients modulo M which is not necessary of kind $c \cdot 2^k + 1$. Coefficients in multiplication can be up to $O(M^2 n)$ which usually can't be handled precisely enough by floating point types. To resolve the situation we should consider polynomials as

$$A(x) = A_1(x) + A_2(x) \cdot 2^k$$
$$B(x) = B_1(x) + B_2(x) \cdot 2^k$$

where $2^k \approx \sqrt{M}$. Then coefficients will be $O(\sqrt{M})$ and product can be written as

$$A \cdot B = A_1 B_1 + (A_1 B_2 + A_2 B_1) \cdot 2^k + A_2 B_2 \cdot 2^{2k}$$

Thus coefficients in product will be $O(Mn)$ which can be handled with allowed precision. This product can be calculated in two direct and two inverse Fourier transforms. Alternative approach which considers multiplication with multiple modulo and then using Chinese remainder theorem is harder to write and also is slower in practice.

Multidimensional Fourier transform As for now we worked only with polynomials with single variable. But similar constructions work for polynomials of multiple variables. Now we have to calculate values in points $(w_1^{k_1}, w_2^{k_2}, \ldots, w_m^{k_m})$. Turns out that for such transform one only have to make one-dimensional transform along each axis given that other coordinates are fixed. In two-dimensional case it means to simply make at first one-dimensional transform of all rows and then of all columns.

Let's prove it for two-dimensional case. We want to gather set of numbers

$$P_{uv} = P(w_1^u, w_2^v) = \sum_{i=0}^{n-1} \sum_{j=0}^{m-1} a_{ij} w_1^u w_2^v$$

At first we have table $A_{uv} = a_{uv}$, after the transform of rows we will get

$$A'_{uv} = P_u(w_2^v) = \sum_{j=0}^{m-1} A_{uj} w_2^v = \sum_{j=0}^{m-1} a_{uj} w_2^v$$

And after following transform of columns we will get

$$A''_{uv} = P'_v(w_1^u) = \sum_{i=0}^{n-1} A'_{iv} w_1^u = \sum_{i=0}^{n-1} P_i(w_2^v) w_1^u = \sum_{i=0}^{n-1} \sum_{j=0}^{m-1} w_1^u w_2^v$$

Such transform allows effectively compute convolutions $C(x, y) = A(x, y) \cdot B(x, y)$ which are

$$c_{uv} = \sum_{\substack{i_1 + j_1 = u \\ i_2 + j_2 = v}} \sum a_{i_1 i_2} b_{j_1 j_2}$$

Walsh-Hadamard transform and other convolutions Computing values of multidimensional polynomials in some special points we can compute some different convolutions:

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

Here $|$, $\&$ and $\oplus$ stand for bit-wise *or*, *and* and *xor* respectively.

1. *xor*. Consider values of polynomial in points of hyper-cube $x \in \{-1, 1\}^k$. For such points stands $x_i^a x_i^b = x_i^{a \ xor \ b}$ thus multiplication of values in such points will correspond to the values of polynomial in which monomials are multiplied following this rule.

   In other words if we consider power of x_i in monomial as i^{th} bit of its index then we can assume that after multiplication of two monomials we get the monomial which index is *xor* of indices of initial monomials so product of two polynomials will indeed correspond to the *xor*-convolution.

   Note that this is nothing but computation of multidimensional Fourier transform in roots of unity of power 2. It's also called Walsh-Hadamard. It can be calculated using integers and also $w^{-1} = w = 1$ thus for inverse transform we can only process direct transform one more time and divide everything by n.

   Here is the code which calculates direct transform:

```
void transform(int *from, int *to) {
    if(to - from == 1) {
        return;
    }
    int *mid = from + (to - from) / 2;
    transform(from, mid);
    transform(mid, to);
    for(int i = 0; i < mid - from; i++) {
        int a = *(from + i);
        int b = *(mid + i);
        *(from + i) = a + b;
        *(mid + i) = a - b;
    }
}
```

2. *or*. Now consider values in points $x \in \{0,1\}^k$. For them stands $x_i^a x_i^b = x_i^{a \text{ or } b}$ which follows that product of monomials can be considered to be monomial which index is *or* of initial indices. Note that value of polynomial in some point is the sum over all sub-masks of this index.

```
void transform(int *from, int *to) {
    if(to - from == 1) {
        return;
    }
    int *mid = from + (to - from) / 2;
    transform(from, mid);
    transform(mid, to);
    for(int i = 0; i < mid - from; i++) {
        *(mid + i) += *(from + i);
    }
}

void inverse(int *from, int *to) {
    if(to - from == 1) {
        return;
    }
    int *mid = from + (to - from) / 2;
    inverse(from, mid);
    inverse(mid, to);
    for(int i = 0; i < mid - from; i++) {
        *(mid + i) -= *(from + i);
    }
}
```

3. *and*. To calculate this convolution, we have to either change all masks to their complement and then calculate *or*-convolution or use the idea from previous point and make a summation over all super-masks instead of sub-masks. This will stand for values of polynomial in same points but with implicit re-numeration of indices changed to complement.

Note that these ideas can be generalized to the case when numbers presented in base-d system for $d > 2$ to calculate convolutions with digit-vise sum modulo d or max or min.

Newton method for functions of polynomials We want to solve equation $f(x) = 0$. $f(x)$ can be presented as $f(x) = f(x_0) + f'(x_0)\Delta x + O(\Delta x^2)$. Let's gradually approach to the zero of this function approximating it by linear $g(x_{n+1}) = f(x_n) + f'(x_n)(x_{n+1} - x_n)$ on each step. Solving $g(x_{n+1}) = 0$ we come to

$$x_{n+1} = x_n - \frac{f(x_n)}{f'(x_n)}$$

Which gives $f(x_{n+1}) = O((x_{n+1} - x_n)^2) = O\left(\frac{f(x_n)^2}{f'(x_n)^2}\right)$. In case of invertible derivative it is $O(f(x_n)^2)$. If x is polynomial then using this approach we can double the number of precisely known coefficients on each step. Most common functions of polynomials:

1. Inverse series. One have to solve $PQ = 1 \Rightarrow f(P) = Q - P^{-1}$ so $P_{n+1} = P_n - \frac{Q - P_n^{-1}}{P_n^{-2}} = P_n(2 - QP_n)$.

2. Exponent. $Q = \ln P \Rightarrow f(P) = Q - \ln P$ and $P_{n+1} = P_n - \frac{Q - \ln P_n}{-P_n^{-1}} = P_n(1 + Q - \ln P_n)$.

3. k^{th} root. $Q = P^k \Rightarrow f(P) = Q - P^k$ and $P_{n+1} = P_n + \frac{Q - P_n^k}{kP_n^{k-1}} = P_n\left(\frac{k-1}{k} + \frac{Q}{kP_n^k}\right)$.

There is logarithm in expression for exponent. To compute it we should note that $(\log P)' = P'P^{-1}$, which allows us to recover coefficients of positive powers and to calculate constant summand using some built-in functions. Which will give $O(n \log n)$ in total.

Divide and Conquer You're given equation $AX = B$ and you have to calculate n coefficients of X. But you don't have the whole B because it only can be computed after we calculate new term of X (for example, B can depend on X, see example below). Assume we found first $m \approx n/2$ coefficients and want to find next m, then

$$\begin{cases} X = X_1 + x^m X_2, \\ A = A_1 + x^m A_2, \\ B = B_1 + x^m B_2, \\ AX_1 = B_1 + x^m B_2' \end{cases} \implies AX \bmod x^{2m} = B_1 + x^m(A_1 X_2 + B_2') = B_1 + x^m B_2$$

$$A_1 X_2 = B_2 - B_2' \mod x^m$$

Thus if we keep B_2' which have to be subtracted from B_2 we can reduce computation of X_2 to the same problem but of size $\approx n/2$ which gives us $O(n \log^2 n)$ algorithm.

As an example one can calculate exponent this way since

$$e^A = X \implies X' = A'X$$

It is simple as result from Newton's method on practice and yields smaller constant hidden under the O-notation. Note that one can use similar sceme to calculate $X = AB$ where A is given beforehand and B depends on X.

$$AB_1 = X_1 + x^m X_2' \implies AB \bmod x^{2m} = X_1 + x^m(A_1 B_2 + X_2')$$

$$X_2 = A_1 B_2 + X_2' \mod x^m$$

Division and interpolation Finally let's learn how to divide polynomials with remainder and to compute evaluation and interpolation in arbitrary set of points.

1. Modulo division. We have to represent $A(x) = B(x)D(x) + R(x)$, $\deg R(x) < \deg B(x)$. Let $\deg A = n$, $\deg B = m$. Then $\deg D = n - m$. Also given $\deg R < m$ we can see that coefficients near $\{x^k\}_{k=m}^{n}$ don't depend on $R(x)$. Thus we have system of $n - m + 1$ linear equations with $n - m + 1$ variables (coefficients of D).

   Consider $A^r(x) = x^n A\left(x^{-1}\right)$, $B^r(x) = x^m B\left(x^{-1}\right)$, $D^r(x) = x^{n-m} D\left(x^{-1}\right)$ which are same polynomials having coefficients reversed. For $n - m + 1$ leading coefficients given that $P(x) \mod z^k$ to be first k coefficients of $P(x)$ we got system

   $$A^r(x) = B^r(x)D^r(x) \mod z^{n-m+1}$$

   Its solution is $D^r(x) = A^r(x)[B^r(x)]^{-1} \bmod z^{n-m+1}$, which allows us to find $D(x)$ and $R(x)$ from it.

2. Multipoint evaluation. We have to compute $P(x_i)$ for $\{x_i\}_{i=1}^{n}$. Given that $P(x_i) = P \mod (x - x_i)$, let's calculate $P \mod \prod_{i=1}^{n/2-1}(x - x_i)$ and $P \mod \prod_{i=n/2}^{n}(x - x_i)$ and proceed recursively which will give us $O(n \log^2 n)$.

3. Interpolation. You're given set $\{(x_i, y_i)\}_{i=0}^{n-1}$, you have to find $P : P(x_i) = y_i$. Assume we found P_1 for first $n/2$ points. Then $P = P_1 + P_2 \prod_{i=0}^{n/2-1}(x - x_i) = P_1 + P_2 Q$. Let's reduce computation of P_2 to interpolation and multipoint evaluation: $P_2(x_i) = \dfrac{y_i - P_1(x_i)}{Q(x_i)}$ for $i > n/2$ which gives us $O(n \log^3 n)$.

3 Exercises

Knapsack There are n types of objects. i^{th} object has cost s_i. Let $s = \sum_{i=1}^{n} s_i$. Suggest algorithm which finds number of ways to choose subset of objects with total cost exactly w for all $w \leq s$ in $O(s \log s \log n)$.

Power sum You're given k and n. Find $\sum_{m=0}^{n} m^k$ in $O(k \log k)$.

Generalized Cooley-Tukey method Let $n = pq$. Suggest algorithm which reduces DFT of size n to p DFTs of size q in $O(n)$ additional operations.

Arythmetic progressions You're given set of n numbers from 0 to m. Find the amount of arithmetical progressions of length 3 in this set in $O(m \log m)$.

Distance between points You're given n points in rectangle $A \times B$. For each possible pair $(\Delta x, \Delta y)$ find how many are there such pairs of points that difference in x-coordinate between them equals Δx and in y-coordinate it is correspondingly Δy in $O(AB \log AB)$.

Pattern matching You're given two strings s and t composed of letters from Σ and questionmarks. Find all positions i such that if we try to match t with s starting in i then in each position letters of s and t either coincide or at least one of them is questionmark in $O(\Sigma n \log n)$ and in $O(n \log n)$*.

Linear recurrents* Sequence F_n is defined as $F_n = \sum_{i=1}^{k} a_{k-i} F_{n-i}$. You're given $\{a_i\}_{i=0}^{k-1}$ and initial values $\{F_i\}_{i=0}^{k-1}$. Suggest algorithm which computes F_n in $O(k \log k \log n)$.

Polynomial power* You're given $P(x) = \sum_{i=0}^{n} a_i x^i$. You have to find first n coefficients of $P^k(x)$ in $O(n \log n)$.