任意模数 NTT 简要介绍（by Singulet31258)

先简要介绍一下普通 NTT。
NTT 是在系数均为整数时，FFT 的一种替代品。因为 FFT 涉及大量的浮点运算，在数据规模很大的时候会有精度问题。而 NTT 仅涉及整数运算，因此不会有精度问题，且整数运算的效率明显高于浮点运算，因此 NTT 的速度同样普遍高于 FFT。
NTT 本质上就是把 FFT 中的 ω_n 换成了 $g^{\frac{2^m q}{n}}$。在模 p 意义下，g 是原根，q 是奇数，$2^m q = p - 1$。设 $\varphi_n \equiv g^{\frac{2^m q}{n}} (\mathrm{mod} p)$，显然，$\varphi_n$ 与 ω_n 的性质基本完全一致。它们都具有周期性，它们的周期都是 n，在同一周期内的 n 个取值都互不相同，等等。
NTT 仅在多项式次数小于 2^m 时能够得出正确结果，而题目的数据规模一般不超过 $n = 10^6 < 2^{20} = 1048576$，因此，我们一般把 $m \geq 21$ 的模数 p 称作 NTT 模数。
由于原根仅在模数为质数的情况下存在，因此 NTT 模数 $p = 2^m q + 1$ 必须是一个质数。
以下为几个常用的 NTT 模数以及它们的常用原根：
32 位整数范围（int）：
$p = 23068673, q = 11, m = 21, g = 3$
$p = 104857601, q = 25, m = 22, g = 3$
$p = 167772161, q = 5, m = 25, g = 3$
$p = 469762049, q = 7, m = 26, g = 3$
$p = 950009857, q = 453, m = 21, g = 7$
$p = 998244353, q = 119, m = 23, g = 3$
$p = 1004535809, q = 479, m = 21, g = 3$
$p = 2013265921, q = 15, m = 27, g = 31$
unsigned int：
$p = 2281701377, q = 17, m = 27, g = 3$
$p = 3221225473, q = 3, m = 30, g = 5$
64 位整数范围（long long）（参考[NTT中可用素数模数原根表]，这里只列出其中 $p \geq 10^{14}$ 的模数）：
$p = 263882790666241, q = 15, m = 44, g = 7$
$p = 1231453023109121, q = 35, m = 45, g = 3$
$p = 1337006139375617, q = 19, m = 46, g = 3$
$p = 3799912185593857, q = 27, m = 47, g = 5$
$p = 4222124650659841, q = 15, m = 48, g = 19$
$p = 7881299347898369, q = 7, m = 50, g = 6$

$p=315251973915934 73,q=7,m=52,g=3$
$p=180143985094819841,q=5,m=55,g=6$
$p=1945555039024054273,q=27,m=56,g=5$
$p=4179340454199820289,q=29,m=57,g=3$
最后 2 个模数大于 10^{18} 但小于 2^{63}。
注：$10^9+7,10^9+9$ 等数虽然是质数，但不属于 NTT 模数，因为它们的 m 都太小，因此在 NTT 时不要使用这些数当模数！

那么问题来了，如果题目强制要求用非 NTT 模数（比如 10^9+7）当模数呢？应该如何处理？

这时，我们就要用到任意模数的 NTT。
考察题目的值域范围，然后选取几个 NTT 模数（也可以只选 1 个，只要选的数本身够大就行，很多 $OIer$ 喜欢选 3 个 NTT 模数：$469762049,998244353,1004535809$，这样的话就需要做 9 次 NTT），它们的乘积超过了值域范围。用这几个模数分别 NTT，然后用 CRT 合并，得到的就是没有取模过的系数（因为此时模数超出值域范围，系数如果取模过说明它也超出了值域范围，这是不可能的）。然后，再对每个系数用题目要求的模数取模即可。
如果题目没要求取模，那么按上述流程操作，到 CRT 合并的时候终止即可。
如果值域范围实在太大（甚至超出了 128 位整数的范围），这个时候就只能使用高精度来处理了，此时可以使用高精度浮点数进行 FFT，也可以使用高精度整数进行 NTT。
还有一种被称为"拆系数 FFT"的算法，在竞赛通常的数据规模下能够更高效地完成任意模数 NTT，因为它只需要做 4 次 FFT，常数只有 3 模 NTT 的 45% 左右。
拆系数 FFT 的算法流程：选取一个常数 M，通常取 2^{15} 或 $\sqrt{P}$，然后将多项式 $A(x),B(x)$ 拆成 $A_0(x)M+A_1(x),B_0(x)M+B_1(x)$。之后构造多项式 $P(x)=A_0(x)+A_1(x)i,Q(x)=B_0(x)+B_1(x)i$，对 $P(x),Q(x)$ 分别做一次 FFT，得到的系数分别为 p_i,q_i。
设 A_0,A_1,B_0,B_1 做 FFT 后得到的系数分别为 s_i,t_i,u_i,v_i，则 $s_i=\frac{p_i+conj(p_{n-i})}{2},t_i=\frac{p_i-conj(p_{n-i})}{2i},u_i=\frac{q_i+conj(q_{n-i})}{2},v_i=\frac{q_i-conj(q_{n-i})}{2i}$，其中 $conj(a+bi)=a-bi,p_n=p_0,q_n=q_0$，具体证明方法可见：拆系数$FFT$（任意模数$NTT$）。
于是，我们便可以 $O(n)$ 求出 s_i,t_i,u_i,v_i。
接下来，构造多项式

$$S(x)=\sum_{i=0}^{n-1}(u_i+iv_i)s_ix^i,T(x)=\sum_{i=0}^{n-1}(u_i+iv_i)t_ix^i$$

然后对 $S(x),T(x)$ 分别做一次 FFT，由于这 2 个多项式的系数实际上是点值，因此这里的 FFT 是点值转系数的 FFT，设 FFT 之后得到的多项式分别是 $F(x),G(x)$。

那么有：$A_0B_0=Re(F),A_0B_1=Im(F),A_1B_0=Re(G),A_1B_1=Im(G)$。
于是 $AB=A_0B_0M^2+(A_0B_1+A_1B_0)M+A_1B_1$ 就可以计算出来了，然后再对各项系数对题目指定的模数取模即可。
这里 A_0,B_0,A_1,B_1 的系数值域上界分别为 $\frac{U}{M},M$，其中 U 是 A,B 的系数值域上界。显然，取 $M=\sqrt{U}$ 时最优。此时多项式乘法的系数值域上界就从原来的 $O((n+m)U^2)$（一般 $U=10^9,n=m=10^5$，那么值域上界就在 10^{23} 这个量级）缩小至 $O((n+m)U)$（此时值域上界在 10^{14} 这个量级），这样就大大减小了 FFT 的浮点运算的误差对结果的影响，然后再用 long double 进行 FFT（有时甚至可以只用 double），就能得到完全精确的没有取模过的结果，这就是拆系数 FFT 能够得出正确结果的原因。
当然，你也可以直接用 __float128 进行 FFT，直接把 A,B 乘起来。因为 __float128 的精度高达 $2^{-112}\approx 1.926\times 10^{-34}$，它能保证至少 33 位十进制有效数字是精确的，这个精度足以胜任竞赛通常的数据规模下所有的多项式乘法。
或者选一个 __int128 类型的超大 NTT 模数 P，通常 $P\geq 10^{30}$，足以胜任竞赛通常的数据规模下所有的多项式乘法，用这种模数直接把 A,B 乘起来即可。
当然，不推荐使用后两种方法，因为 __float128 和 __int128 的常数都是非常大的，实际效率可能还不如 3 模 NTT。
拆系数 FFT 的参考程序见 GitHub 的 Luogu/P4245.cpp。

扩展内容：
1.Number theoretic transforms to implement fast digital convolution
2.FWT(快速沃尔什变换)零基础详解qaq（ACM/OI）
3.Chirp Z 变换