NTT (general setting)

1. Suppose an input is an array of n integers, to make the transform the n-th primitive root of unity (pru) has to be found. Coefficients of forward and inverse transforms (powers pru) usually are precomputed.
2. Choose a minimum (prime) working modulus M such that 1≤n<M and every input value is in the range [0,M).
3. Find n-th pru.

* Select some integer k≥1 and define N=kn+1 as the working modulus, N≥M,  N is a prime. For any n and M [Dirichlet’s theorem](https://en.wikipedia.org/wiki/Dirichlet's_theorem_on_arithmetic_progressions) guarantees an existence of k and N ≥ M.
* Because N is prime, the multiplicative group of ZN has size φ(N)=N−1=kn. It’s a cyclic group and it must have at least one generator g, which is also a (N−1)-th pru.
* Define ω≡gk mod N.

ωn=gkn=gN−1=gφ(N)≡1 mod  due to [Euler’s theorem](https://en.wikipedia.org/wiki/Euler%27s_theorem).

g is a generator => ωi =gik ≢ 1 for 1≤i<n, because ik < nk=N−1. Hence ω is a n-th pru, as required by the DFT of length n.

* For the prime field ZN, one can either find a generator of the field then derive a n-th pru (as mentioned above), or directly find a n-th pru. Such a field has φ(φ(N))=φ(N−1)=φ(kn) generators but φ(n) n-th prus. The first number is greater than or equal to the second number. So the sampling random candidates in the range [0,N) more likely (or equally likely) succeeds in finding a generator than finding a n-th pru.
* To find a generator, first fully factorize N−1 and collect its set of unique prime factors. A candidate a∈(1,N) is a generator of ZN if and only if {for each p in that set of unique prime factors, a(N−1)/p≢1 mod N }.
* To find a primitive root directly, first fully factorize n and collect its set of prime factors. A candidate a∈(1,N) is a n-th pru in ZN if and only if {an≡1 mod Nan≡1 mod N, and for each pp in that set of unique prime factors, an/p≢1 mod N }.

1. When convolving two vectors (or multiplying 2 polynomials) of length n where each input value (coefficient) is at most m, the upper bound on each output value is m2n. Choosing a minimum working modulus of M=m2n+1 is sufficient to always avoid overflow in the worst case.
2. Although the procedure only describes prime fields for simplicity, it might also be possible to operate on composite rings (e.g. Z100) if a n-th pru exists. This could be useful if a composite modulus is much smaller than a prime modulus of the form N=kn+1 (CRT).
3. Furthermore, it should be possible to operate on prime-power fields such as GF(28). However, this is only useful if the input vector is composed of elements from that field, instead of being plain integers.
4. Computing an NTT requires many modular multiplications. It is possible to apply [Montgomery reductions](https://www.nayuki.io/page/montgomery-reduction-algorithm) (or the less efficient [Barrett reductions](https://www.nayuki.io/page/barrett-reduction-algorithm)) to speed up the modular arithmetic in an NTT.

Montgomery reduction algorithm (MRA)

Montgomery reduction is a technique to speed up back-to-back modular multiplications by transforming the numbers into a special form.

For a single multiplication, Montgomery is inferior to doing the modular multiplication directly. But for a chain of multiplications, such as in modular exponentiation, the input numbers can be transformed into Montgomery form, numerous multiplications may be performed and a single inverse transform moving the result back to standard numbers can be performed at the end.

**Summary**

Steps to compute c = (ab mod n):

1. Choose r∈N such that r>n and gcd(r,n)=1.
2. k= (r(r−1 mod n) – 1) / n.
3. a’ = (ar mod n).  
   b’ = (br mod n).
4. x = a’b’.
5. s = (xk mod r).
6. t=x+sn.
7. u=t / r.
8. c’ =if (u<n) then (u) else (u−n).
9. c = (c’r−1 mod n).

## Detailed algorithm

### Initialization

1. Problem: How to compute many instances of c≡a×b mod n without direct application of mod operation. n is fixed, the compute proceeds with many values of a and b.
2. Choose an r that is greater than n and coprime with n. r is going to be a power of 2, which means n needs to be odd and greater than 3, that’s mostly true since n usually is a prime.

The reason for the selection is to make modulo and division by r ops become inexpensive bit masking and right shifting.

1. Let k = (r(r−1 mod n)-1) / n. (This division is exact.)

The reciprocal r−1 mod n exists because r is coprime with n.

rr−1≡1 mod n, thus rr−1=1+kn for some non-negative integer k.

(r−1 mod n) is pre-computed by the [extended Euclidean algorithm](https://en.wikipedia.org/wiki/Extended_Euclidean_algorithm).

### Outer algorithm

1. Since all arithmetic is done modulo n, all input and output numbers are in the range [0,n). Intermediate results in the algorithm may be larger but not negative.

In this discussion, the distinguish is made between equations on integers (x=y) versus congruences of integers modulo a number (x≡y mod n) and between the use of mod as an arithmetic operator versus its use in congruence relations.

1. First convert the input numbers to Montgomery form:

a’ = (ar mod n), b’ = (br mod n).

First assume the desired output is in Montgomery form:

c’ =(cr mod n) =(abr mod n).

1. Compute the product directly:

a’×b’≡ (ar)(br) ≡ abr2 ≡ c’r mod n.

Note that 0 ≤a’×b’<n2.

1. Now compute the reduction (see discussion below):

c’ =Rdc(a’×b’)=(abr mod n).

1. Finally convert the output back to standard form:

c=(c’r−1 mod n) = Rdc(c’), see below.

### Reduction function (Rdc)

1. The reduction function Rdc:[0,n2)→[0,n) computes Rdc(x)=(xr−1 mod n) in an efficient way.
2. Let s=(xk mod r). (k is defined in the initialization.)

0≤s<r. By the properties of modular arithmetic, x in this expression can be replaced with (x mod r).

1. Let t=x+sn.

0≤x<n2<rn and 0≤sn<rn, thus 0≤t<2rn. (Recall that r>n).

The claim:

t is a multiple of r

Proof:  
x+sn ≡ x + xkn ≡ x(1+ kn) ≡ x(rr−1) ≡ 0 mod r.  
Therefore t=ur for some non-negative integer u.

Also t≡x mod n because adding sn preserves the congruence.

1. Let u=t / r. (This division is exact.)

u ≡ u×1 ≡ u(rr−1) ≡ (ur)r−1 ≡ tr−1 mod n.

Moreover, since t ≡ x mod n, u ≡ xr−1 mod n.

1. If 0≤u<n then return u. Otherwise return u−n.

0≤t<2rn => 0≤u<2n. Hence this simple if-else performs a modulo by n correctly. Furthermore, the result is still equal (and congruent) to xr−1 mod n.

**Algorithm.**

Rdc(x) -> y:

s=(xk mod r) # mask

t=x+sn # integer multiply

u = t / r # right log(r) shift

y = u if u < n else u-n # last modulo op