```cpp
// include modll

namespace number_theoretic_transform {

  // when MOD - 1 = 2^m * a,
  // min_omega = root^a (try 3,5,7,... to get root)
  // min_omega_depth = m
  // mod_half = (MOD + 1) / 2

  modll min_omega;
  int min_omega_depth;
  modll mod_half;

  void make_base(int mode) {
    switch (mode) {
      /*
    case 0:
      MOD = 167772161;
      min_omega = 17;
      min_omega_depth = 25;
      mod_half = 83886081;
      break;
    case 1:
      MOD = 469762049;
      min_omega = 30;
      min_omega_depth = 26;
      mod_half = 234881025;
      break;
    case 2:
      MOD = 1224736769;
      min_omega = 149;
      min_omega_depth = 24;
      mod_half = 612368385;
      break;
      */
    default:
      MOD = 924844033;
      min_omega = 3597;
      min_omega_depth = 21;
      mod_half = 462422017;
    }
  }

  vector<modll> omegas, iomegas;

  inline int bit_reverse(int x, int digit) {
    int ret = digit ? x & 1 : 0;
    Loop(i, digit - 1) { ret <<= 1; x >>= 1; ret |= x & 1; }
    return ret;
  }

  inline void make_omegas(int n) {
    if (omegas.size() != n) {
      omegas.resize(n);
      modll omega = pow(min_omega, (1 << min_omega_depth) / n);
      Loop(i, n) {
        if (i == 0) omegas[i] = 1;
        else omegas[i] = omegas[i - 1] * omega;
      }
    }
  }

  inline void make_iomegas(int n) {
    if (iomegas.size() != n) {
      iomegas.resize(n);
      modll iomega = modll(1) / pow(min_omega, (1 << min_omega_depth) / n);
      Loop(i, n) {
        if (i == 0) iomegas[i] = 1;
        else iomegas[i] = iomegas[i - 1] * iomega;
      }
    }
```

```
72    }
73
74    // a.size() should be 2^digit
75    vector<modll> NTT(const vector<modll> a, int mode = 0) {
76      int n = int(a.size());
77      int digit = int(rndf(log2(n)));
78      vector<modll> ret = a;
79      make_omegas(n);
80      Loop(i, n) {
81        int j = bit_reverse(i, digit);
82        if (j > i) swap(ret[i], ret[j]);
83      }
84      Loop(i, digit) {
85        int j = 0, m = 1 << i, mw = (digit - i - 1);
86        Loop(group_id, n >> (i + 1)) {
87          Loop(k, m) {
88            modll x = ret[j] + omegas[k << mw] * ret[j + m];
89            modll y = ret[j] - omegas[k << mw] * ret[j + m];
90            ret[j] = x; ret[j + m] = y;
91            ++j;
92          }
93          j += m;
94        }
95      }
96      return ret;
97    }
98
99    // f.size() should be 2^digit
100   vector<modll> INTT(const vector<modll>& f, int mode = 0) {
101     int n = int(f.size());
102     int digit = int(rndf(log2(n)));
103     vector<modll> ret = f;
104     make_iomegas(n);
105     Loopr(i, digit) {
106       int j = 0, m = 1 << i, mw = (digit - i - 1);
107       Loop(group_id, n >> (i + 1)) {
108         Loop(k, m) {
109           modll q = (ret[j] + ret[j + m]) * mod_half;
110           modll r = (ret[j] - ret[j + m]) * mod_half * iomegas[k << mw];
111           ret[j] = q; ret[j + m] = r;
112           ++j;
113         }
114         j += m;
115       }
116     }
117     Loop(i, n) {
118       int j = bit_reverse(i, digit);
119       if (j > i) swap(ret[i], ret[j]);
120     }
121     return ret;
122   }
123
124   // a.size() = b.size() should be 2^digit
125   vector<modll> mul_convolution(const vector<modll> &a, const vector<modll> &b) {
126     int n = int(a.size());
127     vector<modll> ret;
128     make_base(0);
129     // Garner's algorithm is unsupported yet
130     vector<modll> g = NTT(a), h = NTT(b);
131     Loop(i, n) g[i] *= h[i];
132     ret = INTT(g);
133     return ret;
134   }
135
136   int legal_size_of(int n) {
137     int ret = 1 << (int)log2(n);
138     if (ret < n) ret <<= 1;
139     return ret;
140   }
141 }
142
```

```
143  using namespace number_theoretic_transform;
```