Contents

1 Binary Trie

```
struct trie {
   bool isleaf;
   trie* child[2];
};

trie* create () {
   trie* t = new trie();
   t->isleaf = false;
   memset(t->child, 0, sizeof t->child);
   return t;
}

void add (trie* root, int n) {
   int p = 0;
   for (int i = 31; ~i; --i) {
      p = (n >> i) & 1;
      if (root->child[p] == NULL) {
         root->child[p] = create();
      }
      root = root->child[p];
   }
}
```

```
void clean (trie* root) {
#ifdef CLEAN
   if (root == NULL) return;
   clean(root->child[0]);
   clean(root->child[1]);
   delete (root);
#endif
}

int maxxor (trie* root, int n) {
   int ans = 0;
   for (int i = 31; ~i; --i) {
      int p = (n >> i) & 1;
      if (root->child[p ^ 1] != NULL) {
         p ^= 1;
      }
      root = root->child[p];
      ans <<= 1;
      ans |= p ^ ((n >> i) & 1);
   }
   return ans;
}
```

2 DSU

```
// dsu implementation
vector<int32_t> par(maxn), siz(maxn);

void make_set(int32_t v) {
   par[v] = v;
   siz[v] = 1;
}

int32_t find_set(int32_t v) {
   if (v == par[v]) return v;
   return par[v] = find_set(par[v]);
}

void union_sets(int32_t a, int32_t b) {
   a = find_set(a);
   b = find_set(b);
   if (a != b) {
      if (siz[a] < siz[b]) swap(a, b);
      par[b] = a;
      siz[a] += siz[b];
   }
}
```

3 Fenwick

```
struct Fenwick {
   int n;
   vector<int> t;
   Fenwick(int n) : n(n), t(n + 1) {}

   // prefix_sum[0..i]
   int query (int i) {
      int s = 0;
      while (i) {
         s += t[i];
         i -= i & (-i);
      }
      return s;
   }

   // increase a[i] by v
   void update (int v, int i) {
      while (i <= n) {
         t[i] += v;
```

```
        i += i & (-i);
      }
   }
};
```

4 FFT

```
namespace fft {

   class cmplx {
      public:
         double a, b;
         cmplx() { a = 0.0, b = 0.0; }
         cmplx(double na, double nb = 0.0) { a = na, b = nb; }
         const cmplx operator+(const cmplx& c) { return cmplx(a + c.a, b + c.b); }
         const cmplx operator-(const cmplx& c) { return cmplx(a - c.a, b - c.b); }
         const cmplx operator*(const cmplx& c) {
            return cmplx(a * c.a - b * c.b, a * c.b + b * c.a);
         }
         double magnitude() { return sqrt(a * a + b * b); }
         void print() { cout << "(" << a << ", " << b << ")\n"; }
   };

   const double PI = acos(-1);

   class fft {
      public:
         vector<cmplx> data, roots;
         vector<int32_t> rev;
         int32_t n, s;

         void setSize(int32_t ns) {
            s = ns;
            n = (1 << s);
            int32_t i, j;
            rev = vector<int32_t>(n);
            data = vector<cmplx>(n);
            roots = vector<cmplx>(n + 1);
            for (i = 0; i < n; ++i) {
               for (j = 0; j < s; ++j) {
                  if (i & (1 << j)) {
                     rev[i] += (1 << (s - j - 1));
                  }
               }
            }
            roots[0] = cmplx(1);
            cmplx mult = cmplx(cos(2 * PI / n), sin(2 * PI / n));
            for (i = 1; i <= n; ++i) {
               roots[i] = roots[i - 1] * mult;
            }
         }

         void bitReverse(vector<cmplx>& arr) {
            vector<cmplx> temp(n);
            int32_t i;
            for (i = 0; i < n; ++i) temp[i] = arr[rev[i]];
            for (i = 0; i < n; ++i) arr[i] = temp[i];
         }

         void transform(bool inverse = false) {
            bitReverse(data);
            int32_t i, j, k;
            for (i = 1; i <= s; ++i) {
               int32_t m = (1 << i), md2 = m >> 1;
               int32_t start = 0, increment = (1 << (s - i));
               if (inverse) {
                  start = n;
                  increment *= -1;
               }
               cmplx t, u;
```

```
            for (k = 0; k < n; k += m) {
               int32_t index = start;
               for (j = k; j < md2 + k; ++j) {
                  t = roots[index] * data[j + md2];
                  index += increment;
                  data[j + md2] = data[j] - t;
                  data[j] = data[j] + t;
               }
            }
         }
         if (inverse) {
            for (int32_t i = 0; i < n; ++i) {
               data[i].a /= n;
               data[i].b /= n;
            }
         }
      }

      static vector<int32_t> convolution(vector<int32_t>& a, vector<int32_t>& b) {
         int32_t alen = a.size();
         int32_t blen = b.size();
         int32_t resn = alen + blen - 1;
         int32_t s = 0, i;
         while ((1 << s) < resn) ++s;
         int32_t n = 1 << s;

         fft pga, pgb;
         pga.setSize(s);
         for (i = 0; i < alen; ++i) pga.data[i] = cmplx(a[i]);
         for (i = alen; i < n; ++i) pga.data[i] = cmplx(0);
         pga.transform();

         pgb.setSize(s);
         for (i = 0; i < blen; ++i) pgb.data[i] = cmplx(b[i]);
         for (i = blen; i < n; ++i) pgb.data[i] = cmplx(0);
         pgb.transform();

         for (i = 0; i < n; ++i) pga.data[i] = pga.data[i] * pgb.data[i];
         pga.transform(true);
         vector<int32_t> result(resn);
         for (i = 0; i < resn; ++i) result[i] = (int32_t)(pga.data[i].a + 0.5);

         int32_t actualSize = resn - 1;
         while (~actualSize && result[actualSize] == 0) --actualSize;
         if (actualSize < 0) actualSize = 0;
         result.resize(actualSize + 1);
         return result;
      }
   };
} // namespace fft
```

5 GCD

```
// use only for non-negative u, v
int gcd(int u, int v) {
   int shift;
   if (u == 0) return v;
   if (v == 0) return u;
   shift = __builtin_ctz(u | v);
   u >>= __builtin_ctz(u);
   do {
      v >>= __builtin_ctz(v);
      if (u > v) {
         swap(u, v);
      }
      v -= u;
   } while (v);
   return u << shift;
}
```

```
// use only for non-negative u, v
long long gcd(long long u, long long v) {
   int shift;
   if (u == 0 || v == 0) return u + v;
   shift = __builtin_ctzll(u | v);
   u >>= __builtin_ctzll(u);
   do {
      v >>= __builtin_ctzll(v);
      if (u > v) {
         swap(u, v);
      }
      v -= u;
   } while (v);
   return u << shift;
}
```

6 Grid

```
// grid functions

int32_t n, m;

bool check(int32_t i, int32_t j) {
   return (i >= 0) && (i < n) && (j >= 0) && (j < m);
}

vector<pair<int32_t, int32_t>> dirs = {{1, 0}, {-1, 0}, {0, 1}, {0, -1}};

string direction = "DURL";
```

7 Hash

```
// custom hash

struct custom_hash {
   // http://xorshift.di.unimi.it/splitmix64.c
   static uint64_t splitmix64(uint64_t x) {
      x += 0x9e3779b97f4a7c15;
      x = (x ^ (x >> 30)) * 0xbf58476d1ce4e5b9;
      x = (x ^ (x >> 27)) * 0x94d049bb133111eb;
      return x ^ (x >> 31);
   }
   size_t operator()(uint64_t x) const {
      static const uint64_t FIXED_RANDOM =
         chrono::steady_clock::now().time_since_epoch().count();
      return splitmix64(x + FIXED_RANDOM);
   }
};

struct pair_hash {
   static uint64_t splitmix64(uint64_t x) {
      x += 0x9e3779b97f4a7c15;
      x = (x ^ (x >> 30)) * 0xbf58476d1ce4e5b9;
      x = (x ^ (x >> 27)) * 0x94d049bb133111eb;
      return x ^ (x >> 31);
   }
   size_t operator()(uint64_t x) const {
      static const uint64_t FIXED_RANDOM =
         chrono::steady_clock::now().time_since_epoch().count();
      return splitmix64(x + FIXED_RANDOM);
   }
   size_t operator()(pair<int, int> p) const {
      static const uint64_t FIXED_RANDOM =
         chrono::steady_clock::now().time_since_epoch().count();
      return splitmix64(p.first * 31 + p.second + FIXED_RANDOM);
   }
};
```

8 O(1) square root

```
// O(1) square root

inline long long isqrt(long long n) {
   double N = n;
   N = sqrtl(N);
   long long sq = N - 2;
   sq = max(sq, 0LL);
   while (sq * sq < n) {
      sq++;
   }
   if ((sq * sq) == n) return sq;
   return sq - 1;
}

// segment tree
```

9 Matrix

```
// matrix library

template <typename T>
struct Matrix {
   int32_t rows, cols;
   vector<vector<T>> mat;
   Matrix(int32_t r, int32_t c)
      : rows(r), cols(c), mat(vector<vector<T>>(r, vector<T>(c))){};
   void fill(T val) {
      for (int32_t i = 0; i < rows; i++) {
         for (int32_t j = 0; j < cols; j++) {
            mat[i][j] = val;
         }
      }
   }
   void reset() { fill(0); }
   void setid() {
      assert(rows == cols);
      for (int32_t i = 0; i < rows; i++) {
         mat[i][i] = 1;
      }
   }
   static Matrix id(int32_t n) {
      Matrix m(n, n);
      m.setid();
      return m;
   }
   Matrix operator+(const Matrix& a) const {
      assert(rows == a.rows && cols == a.cols);
      Matrix<T> res(rows, cols);
      for (int32_t i = 0; i < rows; i++) {
         for (int32_t j = 0; j < cols; j++) {
            res.mat[i][j] = mat[i][j] + a.mat[i][j];
         }
      }
   }
   Matrix<T> operator*(const Matrix<T>& a) const {
      assert(cols == a.rows);
      Matrix<T> res(rows, a.cols);
      for (int32_t i = 0; i < rows; i++) {
         for (int32_t j = 0; j < a.cols; j++) {
            res.mat[i][j] = 0;
            for (int32_t k = 0; k < cols; k++) {
               res.mat[i][j] += mat[i][k] * a.mat[k][j];
            }
         }
      }
      return res;
   }
```

```
    void operator+=(const Matrix& a) { *this = *this + a; }
    void operator*=(const Matrix& a) { *this = *this * a; }
};
```

10 Mint

```
// modular int library

template <int32_t MOD = 998'244'353>
struct Modular {
    int32_t value;
    static const int32_t MOD_value = MOD;

    Modular(long long v = 0) {
        value = v % MOD;
        if (value < 0) value += MOD;
    }
    Modular(long long a, long long b) : value(0) {
        *this += a;
        *this /= b;
    }

    Modular& operator+=(Modular const& b) {
        value += b.value;
        if (value >= MOD) value -= MOD;
        return *this;
    }
    Modular& operator-=(Modular const& b) {
        value -= b.value;
        if (value < 0) value += MOD;
        return *this;
    }
    Modular& operator*=(Modular const& b) {
        value = (long long)value * b.value % MOD;
        return *this;
    }

    friend Modular mexp(Modular a, long long e) {
        Modular res = 1;
        while (e) {
            if (e & 1) res *= a;
            a *= a;
            e >>= 1;
        }
        return res;
    }
    friend Modular inverse(Modular a) { return mexp(a, MOD - 2); }

    Modular& operator/=(Modular const& b) { return *this *= inverse(b); }
    friend Modular operator+(Modular a, Modular const b) { return a += b; }
    friend Modular operator-(Modular a, Modular const b) { return a -= b; }
    friend Modular operator-(Modular const a) { return 0 - a; }
    friend Modular operator*(Modular a, Modular const b) { return a *= b; }
    friend Modular operator/(Modular a, Modular const b) { return a /= b; }
    friend std::ostream& operator<<(std::ostream& os, Modular const& a) {
        return os << a.value;
    }
    friend bool operator==(Modular const& a, Modular const& b) {
        return a.value == b.value;
    }
    friend bool operator!=(Modular const& a, Modular const& b) {
        return a.value != b.value;
    }
};

using mint = Modular<mod>;
```

11 NT

```cpp
template <class T, class F = multiplies<T>>
T power(T a, long long n, F op = multiplies<T>(), T e = {1}) {
   assert(n >= 0);
   T res = e;
   while (n) {
      if (n & 1) res = op(res, a);
      if (n >>= 1) a = op(a, a);
   }
   return res;
}

template <unsigned Mod = 998'244'353>
struct Modular {
   using M = Modular;
   unsigned v;
   Modular(long long a = 0) : v((a %= Mod) < 0 ? a + Mod : a) {}
   M operator-() const { return M() -= *this; }
   M& operator+=(M r) {
      if ((v += r.v) >= Mod) v -= Mod;
      return *this;
   }
   M& operator-=(M r) {
      if ((v += Mod - r.v) >= Mod) v -= Mod;
      return *this;
   }
   M& operator*=(M r) {
      v = (uint64_t)v * r.v % Mod;
      return *this;
   }
   M& operator/=(M r) { return *this *= power(r, Mod - 2); }
   friend M operator+(M l, M r) { return l += r; }
   friend M operator-(M l, M r) { return l -= r; }
   friend M operator*(M l, M r) { return l *= r; }
   friend M operator/(M l, M r) { return l /= r; }
   friend bool operator==(M l, M r) { return l.v == r.v; }
   friend bool operator!=(M l, M r) { return l.v != r.v; }
   friend ostream& operator<<(ostream& os, M& a) { return os << a.v; }
   friend istream& operator>>(istream& is, M& a) {
      int64_t w;
      is >> w;
      a = M(w);
      return is;
   }
};

const unsigned mod = 1e9 + 7;

using mint = Modular<>;

// smallest prime divisor computation

vector<int32_t> spf(maxa, -1);

void precompute() {
   spf[0] = spf[1] = 1;
   for (int32_t i = 2; i < maxa; i++) {
      if (spf[i] == -1) {
         for (int32_t j = i; j < maxa; j += i) {
            if (spf[j] == -1) spf[j] = i;
         }
      }
   }
}

// linear sieve
template <int32_t SZ>
struct Sieve {
   bitset<SZ> isprime;
   vector<int32_t> primes;
   Sieve(int32_t n = SZ - 1) {
      for (int32_t i = 2; i <= n; ++i) {
```

```
        if (!isprime[i]) primes.push_back(i);
        for (auto prime : primes) {
           if (i * prime > n) break;
           isprime[i * prime] = true;
           if (i % prime == 0) break;
        }
     }
  }
};

// segmented sieve using O(sqrt(n)) memory, same complexity, cache optimization
struct Sieve {
  vector<int32_t> pr;
  int32_t total_primes;

  Sieve(int32_t n) {
     const int32_t S = 1 << 15;
     int32_t result = 0;
     vector<char> block(S);
     vector<int32_t> primes;

     int32_t nsqrt = sqrt(n);
     vector<char> is_prime(nsqrt + 1, true);

     for (int32_t i = 2; i <= nsqrt; i++) {
        if (is_prime[i]) {
           primes.push_back(i);
           for (int32_t j = i * i; j <= nsqrt; j += i) is_prime[j] = false;
        }
     }

     for (int32_t k = 0; k * S <= n; k++) {
        fill(block.begin(), block.end(), true);
        int32_t start = k * S;
        for (int32_t p : primes) {
           int32_t start_idx = (start + p - 1) / p;
           int32_t j = max(start_idx, p) * p - start;
           for (; j < S; j += p) block[j] = false;
        }
        if (k == 0) block[0] = block[1] = false;
        for (int32_t i = 0; i < S && start + i <= n; i++) {
           if (block[i]) {
              ++result;
              pr.push_back(start + i);
           }
        }
     }
     total_primes = result;
  }
};

// factorial precomputation

vector<mint> fact(maxn);
void precompute_facts() {
  fact[0] = 1;
  for (int32_t i = 0; i < maxn - 1; i++) {
     fact[i + 1] = fact[i] * (i + 1);
  }
}

mint C(int32_t n, int32_t k) { return fact[n] / (fact[k] * fact[n - k]); }

mint P(int32_t n, int32_t k) { return fact[n] / fact[n - k]; }

// O(1) square root

inline int64_t isqrt(int64_t n) {
  // long double ideally
  double N = n;
  N = sqrtl(N);
```

```
    int64_t sq = N - 2;
    sq = max(sq, 0LL);
    while (sq * sq < n) ++sq;
    if (sq * sq == n) return sq;
    return sq - 1;
}

namespace primeCount {
    // https://en.wikipedia.org/wiki/Prime-counting_function
    const int32_t Maxn = 1e7 + 10;
    const int32_t MaxPrimes = 1e6 + 10;
    const int32_t PhiN = 1e5;
    const int32_t PhiK = 100;

    // uint32_t ar[(Maxn >> 6) + 5] = {0};
    int32_t len = 0; // num of primes
    int32_t primes[MaxPrimes];
    int32_t pi[Maxn];
    int32_t dp[PhiN][PhiK];
    bitset<Maxn> fl;

    void sieve(int32_t n) {
        fl[1] = true;
        for (int32_t i = 4; i <= n; i += 2) fl[i] = true;
        for (int32_t i = 3; i * i <= n; i += 2) {
            if (!fl[i]) {
                for (int32_t j = i * i; j <= n; j += i << 1) fl[j] = true;
            }
        }
        for (int32_t i = 1; i <= n; i++) {
            if (!fl[i]) primes[len++] = i;
            pi[i] = len;
        }
    }

    void init() {
        sieve(Maxn - 1);
        int32_t n, k, res;
        for (n = 0; n < PhiN; ++n) dp[n][0] = n;
        for (k = 1; k < PhiK; ++k) {
            int32_t p = primes[k - 1];
            for (n = 0; n < PhiN; ++n) {
                dp[n][k] = dp[n][k - 1] - dp[n / p][k - 1];
            }
        }
    }

    // for sum of primes, multiply the subtracted term by primes[k - 1] in both this
    // and dp
    int64_t non_multiples(int64_t n, int32_t k) {
        if (n < PhiN && k < PhiK) return dp[n][k];
        if (k == 1) return ((++n) >> 1);
        if (primes[k - 1] >= n) return 1;
        return non_multiples(n, k - 1) - non_multiples(n / primes[k - 1], k - 1);
    }

    int64_t Legendre(int64_t n) {
        if (n < Maxn) return pi[n];
        int32_t lim = sqrt(n) + 1;
        int32_t k = upper_bound(primes, primes + len, lim) - primes;
        return non_multiples(n, k) + k - 1;
    }

    // complexity: n^(2/3) (log n^(1/3))
    // Lehmer's method to calculate pi(n)
    int64_t Lehmer(int64_t n) {
        if (n < Maxn) return pi[n];
        int64_t w, res = 0;
        int32_t i, j, a, b, c, lim;
        b = sqrt(n), c = cbrt(n), a = Lehmer(sqrt(b)), b = Lehmer(b);
        res = non_multiples(n, a) + (((b + a - 2) * (b - a + 1)) >> 1);
```

```
      for (i = a; i < b; ++i) {
         w = n / primes[i];
         lim = Lehmer(sqrt(w)), res -= Lehmer(w);
         if (i <= c) {
            for (j = i; j < lim; ++j) {
               res += j;
               res -= Lehmer(w / primes[j]);
            }
         }
      }
      return res;
   }
} // namespace primeCount

namespace fastPrimeCount {
   inline int64_t isqrt(int64_t n) {
      // long double ideally
      double N = n;
      N = sqrtl(N);
      int64_t sq = N - 2;
      sq = max(sq, (int64_t)0);
      while (sq * sq < n) ++sq;
      if (sq * sq == n) return sq;
      return sq - 1;
   }
   int64_t prime_pi(const int64_t N) {
      if (N <= 1) return 0;
      if (N == 2) return 1;
      const int32_t v = isqrt(N);
      int32_t s = (v + 1) / 2;
      vector<int32_t> smalls(s);
      for (int32_t i = 1; i < s; ++i) smalls[i] = i;
      vector<int32_t> roughs(s);
      for (int32_t i = 0; i < s; ++i) roughs[i] = 2 * i + 1;
      vector<int64_t> larges(s);
      for (int32_t i = 0; i < s; ++i) larges[i] = (N / (2 * i + 1) - 1) / 2;
      vector<bool> skip(v + 1);
      const auto divide = [](int64_t n, int64_t d) -> int32_t {
         return (double)(n) / d;
      };
      const auto half = [](int32_t n) -> int32_t { return (n - 1) >> 1; };
      int32_t pc = 0;
      for (int32_t p = 3; p <= v; p += 2)
         if (!skip[p]) {
            int32_t q = p * p;
            if (int64_t(q) * q > N) break;
            skip[p] = true;
            for (int32_t i = q; i <= v; i += 2 * p) skip[i] = true;
            int32_t ns = 0;
            for (int32_t k = 0; k < s; ++k) {
               int32_t i = roughs[k];
               if (skip[i]) continue;
               int64_t d = int64_t(i) * p;
               larges[ns] = larges[k] -
                  (d <= v ? larges[smalls[d >> 1] - pc]
                   : smalls[half(divide(N, d))]) +
                  pc;
               roughs[ns++] = i;
            }
            s = ns;
            for (int32_t i = half(v), j = ((v / p) - 1) | 1; j >= p; j -= 2) {
               int32_t c = smalls[j >> 1] - pc;
               for (int32_t e = (j * p) >> 1; i >= e; --i) smalls[i] -= c;
            }
            ++pc;
         }
      larges[0] += int64_t(s + 2 * (pc - 1)) * (s - 1) / 2;
      for (int32_t k = 1; k < s; ++k) larges[0] -= larges[k];
      for (int32_t l = 1; l < s; ++l) {
         int32_t q = roughs[l];
         int64_t M = N / q;
```

```
        int32_t e = smalls[half(M / q)] - pc;
        if (e < l + 1) break;
        int64_t t = 0;
        for (int32_t k = l + 1; k <= e; ++k)
          t += smalls[half(divide(M, roughs[k]))];
        larges[0] += t - int64_t(e - l) * (pc + l - 1);
      }
      return larges[0] + 1;
   }
} // namespace fastPrimeCount
```

12 NTT

```
#include <bits/stdc++.h>
using namespace std;

template <class T, class F = multiplies<T>>
T power(T a, long long n, F op = multiplies<T>(), T e = {1}) {
   assert(n >= 0);
   T res = e;
   while (n) {
      if (n & 1) res = op(res, a);
      if (n >>= 1) a = op(a, a);
   }
   return res;
}

template <unsigned Mod = 998'244'353>
struct Modular {
   using M = Modular;
   unsigned v;
   Modular(long long a = 0) : v((a %= Mod) < 0 ? a + Mod : a) {}
   M operator-() const { return M() -= *this; }
   M& operator+=(M r) {
      if ((v += r.v) >= Mod) v -= Mod;
      return *this;
   }
   M& operator-=(M r) {
      if ((v += Mod - r.v) >= Mod) v -= Mod;
      return *this;
   }
   M& operator*=(M r) {
      v = (uint64_t)v * r.v % Mod;
      return *this;
   }
   M& operator/=(M r) { return *this *= power(r, Mod - 2); }
   friend M operator+(M l, M r) { return l += r; }
   friend M operator-(M l, M r) { return l -= r; }
   friend M operator*(M l, M r) { return l *= r; }
   friend M operator/(M l, M r) { return l /= r; }
   friend bool operator==(M l, M r) { return l.v == r.v; }
   friend bool operator!=(M l, M r) { return l.v != r.v; }
   friend ostream& operator<<(ostream& os, M a) { return os << a.v; }
   friend istream& operator>>(istream& is, M& a) {
      int64_t w;
      is >> w;
      a = M(w);
      return is;
   }
};

using mint = Modular<998244353>;

namespace ntt {

   template <unsigned Mod>
      void ntt(vector<Modular<Mod>>& a, bool inverse) {
         static vector<Modular<Mod>> dw(30), idw(30);
         if (dw[0] == 0) {
            Modular<Mod> root = 2;
```

```
            while (power(root, (Mod - 1) / 2) == 1) root += 1;
            for (int32_t i = 0; i < 30; ++i)
                dw[i] = -power(root, (Mod - 1) >> (i + 2)), idw[i] = 1 / dw[i];
        }
        int32_t n = a.size();
        assert((n & (n - 1)) == 0);
        if (not inverse) {
            for (int32_t m = n; m >>= 1;) {
                Modular<Mod> w = 1;
                int32_t m2 = m << 1;
                for (int32_t s = 0, k = 0; s < n; s += m2) {
                    for (int32_t i = s, j = s + m; i < s + m; ++i, ++j) {
                        auto x = a[i], y = a[j] * w;
                        if (x.v >= Mod) x.v -= Mod;
                        a[i].v = x.v + y.v, a[j].v = x.v + (Mod - y.v);
                        // here a[i] is not normalised
                    }
                    w *= dw[__builtin_ctz(++k)];
                }
            }
        } else {
            for (int32_t m = 1; m < n; m <<= 1) {
                Modular<Mod> w = 1;
                int32_t m2 = m << 1;
                for (int32_t s = 0, k = 0; s < n; s += m2) {
                    for (int32_t i = s, j = s + m; i < s + m; ++i, ++j) {
                        auto x = a[i], y = a[j];
                        a[i] = x + y, a[j].v = x.v + (Mod - y.v), a[j] *= w;
                    }
                    w *= idw[__builtin_ctz(++k)];
                }
            }
            auto inv = 1 / Modular<Mod>(n);
            for (auto&& e : a) e *= inv;
        }
    }
template <unsigned Mod>
    vector<Modular<Mod>> operator*(vector<Modular<Mod>> l, vector<Modular<Mod>> r) {
        if (l.empty() or r.empty()) return {};
        int32_t n = l.size(), m = r.size(), sz = 1 << __lg(((n + m - 1) << 1) - 1);
        if (min(n, m) < 30) {
            vector<long long> res(n + m - 1);
            for (int32_t i = 0; i < n; ++i)
                for (int32_t j = 0; j < m; ++j) res[i + j] += (l[i] * r[j]).v;
            return {begin(res), end(res)};
        }
        bool eq = l == r;
        l.resize(sz), ntt(l, false);
        if (eq)
            r = l;
        else
            r.resize(sz), ntt(r, false);
        for (int32_t i = 0; i < sz; ++i) l[i] *= r[i];
        ntt(l, true), l.resize(n + m - 1);
        return l;
    }

// for 1e9+7 ntt
constexpr long long mod = 1e9 + 7;
using Mint197 = Modular<mod>;

vector<Mint197> operator*(const vector<Mint197>& l, const vector<Mint197>& r) {
    if (l.empty() or r.empty()) return {};
    int n = l.size(), m = r.size();
    static constexpr int mod0 = 998244353, mod1 = 1300234241, mod2 = 1484783617;
    using Mint0 = Modular<mod0>;
    using Mint1 = Modular<mod1>;
    using Mint2 = Modular<mod2>;
    vector<Mint0> l0(n), r0(m);
    vector<Mint1> l1(n), r1(m);
    vector<Mint2> l2(n), r2(m);
```

```cpp
        for (int i = 0; i < n; ++i) l0[i] = l[i].v, l1[i] = l[i].v, l2[i] = l[i].v;
        for (int j = 0; j < m; ++j) r0[j] = r[j].v, r1[j] = r[j].v, r2[j] = r[j].v;
        l0 = l0 * r0, l1 = l1 * r1, l2 = l2 * r2;
        vector<Mint197> res(n + m - 1);
        static const Mint1 im0 = 1 / Mint1(mod0);
        static const Mint2 im1 = 1 / Mint2(mod1), im0m1 = im1 / mod0;
        static const Mint197 m0 = mod0, m0m1 = m0 * mod1;
        for (int i = 0; i < n + m - 1; ++i) {
            int y0 = l0[i].v;
            int y1 = (im0 * (l1[i] - y0)).v;
            int y2 = (im0m1 * (l2[i] - y0) - im1 * y1).v;
            res[i] = y0 + m0 * y1 + m0m1 * y2;
        }
        return res;
    }

} // namespace ntt

using namespace ntt;

namespace IO {
    const int BUFFER_SIZE = 1 << 15;
    char input_buffer[BUFFER_SIZE];
    int input_pos = 0, input_len = 0;
    char output_buffer[BUFFER_SIZE];
    int output_pos = 0;
    char number_buffer[100];
    uint8_t lookup[100];
    void _update_input_buffer() {
        input_len = fread(input_buffer, sizeof(char), BUFFER_SIZE, stdin);
        input_pos = 0;
        if (input_len == 0) input_buffer[0] = EOF;
    }
    inline char next_char(bool advance = true) {
        if (input_pos >= input_len) _update_input_buffer();

        return input_buffer[advance ? input_pos++ : input_pos];
    }

    template <typename T>
        inline void read_int(T& number) {
            bool negative = false;
            number = 0;

            while (!isdigit(next_char(false)))
                if (next_char() == '-') negative = true;

            do {
                number = 10 * number + (next_char() - '0');
            } while (isdigit(next_char(false)));

            if (negative) number = -number;
        }

    template <typename T, typename... Args>
        inline void read_int(T& number, Args&... args) {
            read_int(number);
            read_int(args...);
        }

    void _flush_output() {
        fwrite(output_buffer, sizeof(char), output_pos, stdout);
        output_pos = 0;
    }

    inline void write_char(char c) {
        if (output_pos == BUFFER_SIZE) _flush_output();

        output_buffer[output_pos++] = c;
    }
```

```
    template <typename T>
        inline void write_int(T number, char after = '\0') {
            if (number < 0) {
                write_char('-');
                number = -number;
            }
            int length = 0;
            while (number >= 10) {
                uint8_t lookup_value = lookup[number % 100];
                number /= 100;
                number_buffer[length++] = (lookup_value & 15) + '0';
                number_buffer[length++] = (lookup_value >> 4) + '0';
            }
            if (number != 0 || length == 0) write_char(number + '0');
            for (int i = length - 1; i >= 0; i--) write_char(number_buffer[i]);
            if (after) write_char(after);
        }
    void IOinit() {
        // Make sure _flush_output() is called at the end of the program.
        bool exit_success = atexit(_flush_output) == 0;
        assert(exit_success);
        for (int i = 0; i < 100; i++) lookup[i] = (i / 10 << 4) + i % 10;
    }
} // namespace IO

using namespace IO;

int32_t main() {
    IOinit();
    int n, m;
    read_int(n, m);
    vector<Mint197> a(n), b(m);
    for (auto& x : a) read_int(x);
    for (auto& x : b) read_int(x);
    a = a * b;
    for (int32_t i = 0; i < n + m - 1; ++i) {
        write_int(a[i].v, ' ');
    }
}
```

13 Polynomial

```
namespace algebra {
    const int32_t inf = 1e9;
    const int32_t magic = 500; // threshold for sizes to run the naive algo

    namespace fft {
        const int32_t maxn = 1 << 18;

        typedef double ftype;
        typedef complex<ftype> point;

        point32_t w[maxn];
        const ftype pi = acos(-1);
        bool initiated = 0;
        void init() {
            if (!initiated) {
                for (int32_t i = 1; i < maxn; i *= 2) {
                    for (int32_t j = 0; j < i; j++) {
                        w[i + j] = polar(ftype(1), pi * j / i);
                    }
                }
                initiated = 1;
            }
        }
        template <typename T>
            void fft(T* in, point32_t* out, int32_t n, int32_t k = 1) {
                if (n == 1) {
                    *out = *in;
                } else {
```

```
            n /= 2;
            fft(in, out, n, 2 * k);
            fft(in + k, out + n, n, 2 * k);
            for (int32_t i = 0; i < n; i++) {
               auto t = out[i + n] * w[i + n];
               out[i + n] = out[i] - t;
               out[i] += t;
            }
         }
      }

   template <typename T>
      void mul_slow(vector<T>& a, const vector<T>& b) {
         vector<T> res(a.size() + b.size() - 1);
         for (size_t i = 0; i < a.size(); i++) {
            for (size_t j = 0; j < b.size(); j++) {
               res[i + j] += a[i] * b[j];
            }
         }
         a = res;
      }

   template <typename T>
      void mul(vector<T>& a, const vector<T>& b) {
         if (min(a.size(), b.size()) < magic) {
            mul_slow(a, b);
            return;
         }
         init();
         static const int32_t shift = 15, mask = (1 << shift) - 1;
         size_t n = a.size() + b.size() - 1;
         while (__builtin_popcount(n) != 1) {
            n++;
         }
         a.resize(n);
         static point32_t A[maxn], B[maxn];
         static point32_t C[maxn], D[maxn];
         for (size_t i = 0; i < n; i++) {
            A[i] = point(a[i] & mask, a[i] >> shift);
            if (i < b.size()) {
               B[i] = point(b[i] & mask, b[i] >> shift);
            } else {
               B[i] = 0;
            }
         }
         fft(A, C, n);
         fft(B, D, n);
         for (size_t i = 0; i < n; i++) {
            point32_t c0 = C[i] + conj(C[(n - i) % n]);
            point32_t c1 = C[i] - conj(C[(n - i) % n]);
            point32_t d0 = D[i] + conj(D[(n - i) % n]);
            point32_t d1 = D[i] - conj(D[(n - i) % n]);
            A[i] = c0 * d0 - point(0, 1) * c1 * d1;
            B[i] = c0 * d1 + d0 * c1;
         }
         fft(A, C, n);
         fft(B, D, n);
         reverse(C + 1, C + n);
         reverse(D + 1, D + n);
         int32_t t = 4 * n;
         for (size_t i = 0; i < n; i++) {
            int64_t A0 = llround(real(C[i]) / t);
            T A1 = llround(imag(D[i]) / t);
            T A2 = llround(imag(C[i]) / t);
            a[i] = A0 + (A1 << shift) + (A2 << 2 * shift);
         }
         return;
      }
} // namespace fft
template <typename T>
   T bpow(T x, size_t n) {
```

```
        return n ? n % 2 ? x * bpow(x, n - 1) : bpow(x * x, n / 2) : T(1);
    }
template <typename T>
    T bpow(T x, size_t n, T m) {
        return n ? n % 2 ? x * bpow(x, n - 1, m) % m : bpow(x * x % m, n / 2, m)
            : T(1);
    }
template <typename T>
    T gcd(const T& a, const T& b) {
        return b == T(0) ? a : gcd(b, a % b);
    }
template <typename T>
    T nCr(T n, int32_t r) { // runs in O(r)
        T res(1);
        for (int32_t i = 0; i < r; i++) {
            res *= (n - T(i));
            res /= (i + 1);
        }
        return res;
    }

template <int32_t m>
    struct modular {
        int64_t r;
        modular() : r(0) {}
        modular(int64_t rr) : r(rr) {
            if (abs(r) >= m) r %= m;
            if (r < 0) r += m;
        }
        modular inv() const { return bpow(*this, m - 2); }
        modular operator*(const modular& t) const { return (r * t.r) % m; }
        modular operator/(const modular& t) const { return *this * t.inv(); }
        modular operator+=(const modular& t) {
            r += t.r;
            if (r >= m) r -= m;
            return *this;
        }
        modular operator-=(const modular& t) {
            r -= t.r;
            if (r < 0) r += m;
            return *this;
        }
        modular operator+(const modular& t) const { return modular(*this) += t; }
        modular operator-(const modular& t) const { return modular(*this) -= t; }
        modular operator*=(const modular& t) { return *this = *this * t; }
        modular operator/=(const modular& t) { return *this = *this / t; }

        bool operator==(const modular& t) const { return r == t.r; }
        bool operator!=(const modular& t) const { return r != t.r; }

        operator int64_t() const { return r; }
    };
template <int32_t T>
    istream& operator>>(istream& in, modular<T>& x) {
        return in >> x.r;
    }

template <typename T>
    struct poly {
        vector<T> a;

        void normalize() { // get rid of leading zeroes
            while (!a.empty() && a.back() == T(0)) {
                a.pop_back();
            }
        }

        poly() {}
        poly(T a0) : a{a0} { normalize(); }
        poly(vector<T> t) : a(t) { normalize(); }
```

```
    poly operator+=(const poly& t) {
        a.resize(max(a.size(), t.a.size()));
        for (size_t i = 0; i < t.a.size(); i++) {
            a[i] += t.a[i];
        }
        normalize();
        return *this;
    }
    poly operator-=(const poly& t) {
        a.resize(max(a.size(), t.a.size()));
        for (size_t i = 0; i < t.a.size(); i++) {
            a[i] -= t.a[i];
        }
        normalize();
        return *this;
    }
    poly operator+(const poly& t) const { return poly(*this) += t; }
    poly operator-(const poly& t) const { return poly(*this) -= t; }

    poly mod_xk(size_t k) const { // get same polynomial mod x^k
        k = min(k, a.size());
        return vector<T>(begin(a), begin(a) + k);
    }
    poly mul_xk(size_t k) const { // multiply by x^k
        poly res(*this);
        res.a.insert(begin(res.a), k, 0);
        return res;
    }
    poly div_xk(size_t k) const { // divide by x^k, dropping coefficients
        k = min(k, a.size());
        return vector<T>(begin(a) + k, end(a));
    }
    poly substr(size_t l, size_t r) const { // return mod_xk(r).div_xk(l)
        l = min(l, a.size());
        r = min(r, a.size());
        return vector<T>(begin(a) + l, begin(a) + r);
    }
    poly inv(size_t n) const { // get inverse series mod x^n
        assert(!is_zero());
        poly ans = a[0].inv();
        size_t a = 1;
        while (a < n) {
            poly C = (ans * mod_xk(2 * a)).substr(a, 2 * a);
            ans -= (ans * C).mod_xk(a).mul_xk(a);
            a *= 2;
        }
        return ans.mod_xk(n);
    }

    // change fft to ntt if using modulo
    poly operator*=(const poly& t) {
        fft::mul(a, t.a);
        normalize();
        return *this;
    }
    poly operator*(const poly& t) const { return poly(*this) *= t; }

    poly reverse(size_t n,
            bool rev = 0) const { // reverses and leaves only n terms
        poly res(*this);
        if (rev) { // If rev = 1 then tail goes to head
            res.a.resize(max(n, res.a.size()));
        }
        std::reverse(res.a.begin(), res.a.end());
        return res.mod_xk(n);
    }

    pair<poly, poly> divmod_slow(
            const poly& b) const { // when divisor or quotient is small
        vector<T> A(a);
        vector<T> res;
```

```
    while (A.size() >= b.a.size()) {
        res.push_back(A.back() / b.a.back());
        if (res.back() != T(0)) {
            for (size_t i = 0; i < b.a.size(); i++) {
                A[A.size() - i - 1] -= res.back() * b.a[b.a.size() - i - 1];
            }
        }
        A.pop_back();
    }
    std::reverse(begin(res), end(res));
    return {res, A};
}

pair<poly, poly> divmod(
        const poly& b) const { // returns quotiend and remainder of a mod b
    if (deg() < b.deg()) {
        return {poly{0}, *this};
    }
    int32_t d = deg() - b.deg();
    if (min(d, b.deg()) < magic) {
        return divmod_slow(b);
    }
    poly D = (reverse(d + 1) * b.reverse(d + 1).inv(d + 1))
        .mod_xk(d + 1)
        .reverse(d + 1, 1);
    return {D, *this - D * b};
}

poly operator/(const poly& t) const { return divmod(t).first; }
poly operator%(const poly& t) const { return divmod(t).second; }
poly operator/=(const poly& t) { return *this = divmod(t).first; }
poly operator%=(const poly& t) { return *this = divmod(t).second; }
poly operator*=(const T& x) {
    for (auto& it : a) {
        it *= x;
    }
    normalize();
    return *this;
}
poly operator/=(const T& x) {
    for (auto& it : a) {
        it /= x;
    }
    normalize();
    return *this;
}
poly operator*(const T& x) const { return poly(*this) *= x; }
poly operator/(const T& x) const { return poly(*this) /= x; }

void print() const {
    for (auto it : a) {
        cout << it << ' ';
    }
    cout << endl;
}
T eval(T x) const { // evaluates in single point x
    T res(0);
    for (int32_t i = int32_t(a.size()) - 1; i >= 0; i--) {
        res *= x;
        res += a[i];
    }
    return res;
}

T& lead() { // leading coefficient
    return a.back();
}
int32_t deg() const { // degree
    return a.empty() ? -inf : a.size() - 1;
}
bool is_zero() const { // is polynomial zero
```

```
        return a.empty();
    }
    T operator[](int32_t idx) const {
        return idx >= (int)a.size() || idx < 0 ? T(0) : a[idx];
    }

    T& coef(size_t idx) { // mutable reference at coefficient
        return a[idx];
    }
    bool operator==(const poly& t) const { return a == t.a; }
    bool operator!=(const poly& t) const { return a != t.a; }

    poly deriv() { // calculate derivative
        vector<T> res;
        for (int32_t i = 1; i <= deg(); i++) {
            res.push_back(T(i) * a[i]);
        }
        return res;
    }
    poly integr() { // calculate integral with C = 0
        vector<T> res = {0};
        for (int32_t i = 0; i <= deg(); i++) {
            res.push_back(a[i] / T(i + 1));
        }
        return res;
    }
    size_t leading_xk() const { // Let p(x) = x^k * t(x), return k
        if (is_zero()) {
            return inf;
        }
        int32_t res = 0;
        while (a[res] == T(0)) {
            res++;
        }
        return res;
    }
    poly log(size_t n) { // calculate log p(x) mod x^n
        assert(a[0] == T(1));
        return (deriv().mod_xk(n) * inv(n)).integr().mod_xk(n);
    }
    poly exp(size_t n) { // calculate exp p(x) mod x^n
        if (is_zero()) {
            return T(1);
        }
        assert(a[0] == T(0));
        poly ans = T(1);
        size_t a = 1;
        while (a < n) {
            poly C = ans.log(2 * a).div_xk(a) - substr(a, 2 * a);
            ans -= (ans * C).mod_xk(a).mul_xk(a);
            a *= 2;
        }
        return ans.mod_xk(n);
    }
    poly pow_slow(size_t k, size_t n) { // if k is small
        return k ? k % 2 ? (*this * pow_slow(k - 1, n)).mod_xk(n)
            : (*this * *this).mod_xk(n).pow_slow(k / 2, n)
            : T(1);
    }
    poly pow(size_t k, size_t n) { // calculate p^k(n) mod x^n
        if (is_zero()) {
            return *this;
        }
        if (k < magic) {
            return pow_slow(k, n);
        }
        int32_t i = leading_xk();
        T j = a[i];
        poly t = div_xk(i) / j;
        return bpow(j, k) * (t.log(n) * T(k)).exp(n).mul_xk(i * k).mod_xk(n);
    }
```

```
poly mulx(T x) { // component-wise multiplication with x^k
   T cur = 1;
   poly res(*this);
   for (int32_t i = 0; i <= deg(); i++) {
      res.coef(i) *= cur;
      cur *= x;
   }
   return res;
}
poly mulx_sq(T x) { // component-wise multiplication with x^{k^2}
   T cur = x;
   T total = 1;
   T xx = x * x;
   poly res(*this);
   for (int32_t i = 0; i <= deg(); i++) {
      res.coef(i) *= total;
      total *= cur;
      cur *= xx;
   }
   return res;
}
vector<T> chirpz_even(
      T z, int32_t n) { // P(1), P(z^2), P(z^4), ..., P(z^2(n-1))
   int32_t m = deg();
   if (is_zero()) {
      return vector<T>(n, 0);
   }
   vector<T> vv(m + n);
   T zi = z.inv();
   T zz = zi * zi;
   T cur = zi;
   T total = 1;
   for (int32_t i = 0; i <= max(n - 1, m); i++) {
      if (i <= m) {
         vv[m - i] = total;
      }
      if (i < n) {
         vv[m + i] = total;
      }
      total *= cur;
      cur *= zz;
   }
   poly w = (mulx_sq(z) * vv).substr(m, m + n).mulx_sq(z);
   vector<T> res(n);
   for (int32_t i = 0; i < n; i++) {
      res[i] = w[i];
   }
   return res;
}
vector<T> chirpz(T z, int32_t n) { // P(1), P(z), P(z^2), ..., P(z^(n-1))
   auto even = chirpz_even(z, (n + 1) / 2);
   auto odd = mulx(z).chirpz_even(z, n / 2);
   vector<T> ans(n);
   for (int32_t i = 0; i < n / 2; i++) {
      ans[2 * i] = even[i];
      ans[2 * i + 1] = odd[i];
   }
   if (n % 2 == 1) {
      ans[n - 1] = even.back();
   }
   return ans;
}
template <typename iter>
   vector<T> eval(vector<poly>& tree, int32_t v, iter l,
         iter r) { // auxiliary evaluation function
      if (r - l == 1) {
         return {eval(*l)};
      } else {
         auto m = l + (r - l) / 2;
         auto A = (*this % tree[2 * v]).eval(tree, 2 * v, l, m);
         auto B = (*this % tree[2 * v + 1]).eval(tree, 2 * v + 1, m, r);
```

```
                A.insert(end(A), begin(B), end(B));
                return A;
            }
        }
    vector<T> eval(vector<T> x) { // evaluate polynomial in (x1, ..., xn)
        int32_t n = x.size();
        if (is_zero()) {
            return vector<T>(n, T(0));
        }
        vector<poly> tree(4 * n);
        build(tree, 1, begin(x), end(x));
        return eval(tree, 1, begin(x), end(x));
    }
    template <typename iter>
        poly inter(vector<poly>& tree, int32_t v, iter l, iter r, iter ly,
                iter ry) { // auxiliary interpolation function
            if (r - l == 1) {
                return {*ly / a[0]};
            } else {
                auto m = l + (r - l) / 2;
                auto my = ly + (ry - ly) / 2;
                auto A = (*this % tree[2 * v]).inter(tree, 2 * v, l, m, ly, my);
                auto B =
                    (*this % tree[2 * v + 1]).inter(tree, 2 * v + 1, m, r, my, ry);
                return A * tree[2 * v + 1] + B * tree[2 * v];
            }
        }
};
template <typename T>
    poly<T> operator*(const T& a, const poly<T>& b) {
        return b * a;
    }

template <typename T>
    poly<T> xk(int32_t k) { // return x^k
        return poly<T>{1}.mul_xk(k);
    }

template <typename T>
    T resultant(poly<T> a, poly<T> b) { // computes resultant of a and b
        if (b.is_zero()) {
            return 0;
        } else if (b.deg() == 0) {
            return bpow(b.lead(), a.deg());
        } else {
            int32_t pw = a.deg();
            a %= b;
            pw -= a.deg();
            T mul = bpow(b.lead(), pw) * T((b.deg() & a.deg() & 1) ? -1 : 1);
            T ans = resultant(b, a);
            return ans * mul;
        }
    }
template <typename iter>
    poly<typename iter::value_type> kmul(
            iter L, iter R) { // computes (x-a1)(x-a2)...(x-an) without building tree
        if (R - L == 1) {
            return vector<typename iter::value_type>{-*L, 1};
        } else {
            iter M = L + (R - L) / 2;
            return kmul(L, M) * kmul(M, R);
        }
    }
template <typename T, typename iter>
    poly<T> build(vector<poly<T>>& res, int32_t v, iter L,
            iter R) { // builds evaluation tree for (x-a1)(x-a2)...(x-an)
        if (R - L == 1) {
            return res[v] = vector<T>{-*L, 1};
        } else {
            iter M = L + (R - L) / 2;
            return res[v] = build(res, 2 * v, L, M) * build(res, 2 * v + 1, M, R);
```

```
      }
    }
  template <typename T>
    poly<T> inter(
        vector<T> x,
        vector<T> y) { // interpolates minimum polynomial from (xi, yi) pairs
      int32_t n = x.size();
      vector<poly<T>> tree(4 * n);
      return build(tree, 1, begin(x), end(x))
        .deriv()
        .inter(tree, 1, begin(x), end(x), begin(y), end(y));
    }
}; // namespace algebra

using namespace algebra;

const int32_t mod = 1e9 + 7;
typedef modular<mod> base;
typedef poly<base> polyn;
```

14 Removable priority queue

```
// note that this is slower than the usual priority queue if you're using
// dijkstra's algorithm, so only use if the memory O(m) is a bit too big for
// your purposes (like making an implicitly defined graph)

template <typename T, typename V = vector<T>, class S = less<T>>
class custom_priority_queue : public std::priority_queue<T, V, S> {
  public:
    bool remove(const T& value) {
      auto it = std::find(this->c.begin(), this->c.end(), value);
      if (it != this->c.end()) {
        this->c.erase(it);
        std::make_heap(this->c.begin(), this->c.end(), this->comp);
        return true;
      } else {
        return false;
      }
    }
};
```

15 Segment Tree

```
struct SegTree {
  // datatype of nodes of segment tree
  typedef int T;
  // datatype of vector that's generating the segment tree
  typedef int S;
  // identity element of monoid
  // if you have any issues with unit, define it outside the struct as a
  // normal variable
  static constexpr T unit = 0;
  // node of segment tree from a value
  T make_node(S val) { return val; }
  // combine function - needs to be an associative function
  T combine(T a, T b) { return a + b; }
  // point update function - updating the element in the array
  void update_val(T& a, S b) { a += b; }

  vector<T> t;
  int32_t n;

  SegTree(int32_t n = 0, T def = unit) : t(n << 1, def), n(n) {}
  SegTree(vector<S>& a, T def = unit) {
    n = a.size();
    t.assign(n << 1, unit);
    for (int32_t i = 0; i < n; ++i) {
      t[i + n] = make_node(a[i]);
    }
```

```
    for (int32_t i = n - 1; i; --i) {
      t[i] = combine(t[i << 1], t[i << 1 | 1]);
    }
  }

  void update(int32_t pos, S val) {
    for (update_val(t[pos += n], val); pos >>= 1;) {
      t[pos] = combine(t[pos << 1], t[pos << 1 | 1]);
    }
  }

  T query(int32_t l, int32_t r) {
    T ra = unit, rb = unit;
    for (l += n, r += n; l < r; l >>= 1, r >>= 1) {
      if (l & 1) ra = combine(ra, t[l++]);
      if (r & 1) rb = combine(t[--r], rb);
    }
    return combine(ra, rb);
  }
};

typedef struct {
  // datatype
} node;

const node ID;
node t[4 * maxn];

node combine(node n1, node n2) {
  node ans;
  // do something
  return ans;
}

node make_node(int val) {
  // make a node from val and return
}

// build segtree - 1 indexed
void build(int v, int l, int r, vector<int>& a) {
  if (l == r) {
    t[v] = make_node(a[l]);
    return;
  }
  int mid = (l + r) >> 1;
  build(v << 1, l, mid, a);
  build((v << 1) | 1, mid + 1, r, a);
  t[v] = combine(t[(v << 1)], t[(v << 1) | 1]);
}

// update segtree by updating value to val at idx
void update(int v, int l, int r, int idx, int val) {
  if (l == r) {
    t[v] = make_node(val);
    return;
  }
  int mid = (l + r) >> 1;
  if (idx <= mid)
    update(v << 1, l, mid, idx, val);
  else
    update((v << 1) | 1, mid + 1, r, idx, val);
  t[v] = combine(t[v << 1], t[(v << 1) | 1]);
}

// range query from l to r both inclusive
node query(int v, int tl, int tr, int l, int r) {
  if (l > r) return ID;
  if (l == tl && r == tr) {
    return t[v];
  }
  int tm = (tl + tr) >> 1;
```

```
   return combine(query(v << 1, tl, tm, l, min(r, tm)),
         query((v << 1) | 1, tm + 1, tr, max(l, tm + 1), r));
}

// slightly more efficient range query
node query2(int v, int tl, int tr, int l, int r) {
   if (l == tl && r == tr)
      return t[v];
   int tm = (tl + tr) >> 1;
   if (l > tm)
      return query((v << 1) | 1, tm + 1, tr, l, r);
   if (tm + 1 > r)
      return query(v << 1, tl, tm, l, r);
   return combine(query(v << 1, tl, tm, l, tm),
         query((v << 1) | 1, tm + 1, tr, tm + 1, r));
}
```

16 Sparse Table

```
// sparse table
const int N = 1e6 + 6;
const int Log = 26;

int sparse_table[Log][N];

void build_sparse_table(vector<int>& a) {
   int n = a.size();
   for (int i = 0; i < n; ++i) {
      sparse_table[0][i] = a[i];
   }
   for (int j = 0; j < Log - 1; ++j) {
      for (int i = 0; i + (2 << j) <= n; ++i) {
         sparse_table[j + 1][i] =
            min(sparse_table[j][i], sparse_table[j][i + (1 << j)]);
      }
   }
}

//[l, r)
int query(int l, int r) {
   int sz = __lg(r - l);
   return min(sparse_table[sz][l], sparse_table[sz][r - (1 << sz)]);
}
```

17 Suffix Array

```
// suffix array

vector<int32_t> sort_cyclic_shifts(string const& s) {
   int32_t n = s.size();
   const int32_t alphabet = 128;
   vector<int32_t> p(n), c(n), cnt(max(alphabet, n), 0);
   // base case : length = 1, so sort by counting sort
   for (int32_t i = 0; i < n; i++) cnt[s[i]]++;
   for (int32_t i = 1; i < alphabet; i++) cnt[i] += cnt[i - 1];
   for (int32_t i = 0; i < n; i++) p[--cnt[s[i]]] = i;
   c[p[0]] = 0;
   int32_t classes = 1;
   for (int32_t i = 1; i < n; i++) {
      if (s[p[i]] != s[p[i - 1]]) ++classes;
      c[p[i]] = classes - 1;
   }
   // inductive case, sort by radix sort on pairs (in fact you only need to
   // sort by first elements now)
   vector<int32_t> p_new(n), c_new(n);
   for (int32_t h = 0; (1 << h) < n; ++h) {
      for (int32_t i = 0; i < n; i++) {
         p_new[i] = p[i] - (1 << h);
         if (p_new[i] < 0) p_new[i] += n;
```

```
      }
      fill(cnt.begin(), cnt.begin() + classes, 0);
      for (int32_t i = 0; i < n; i++) cnt[c[p_new[i]]]++;
      for (int32_t i = 1; i < classes; i++) cnt[i] += cnt[i - 1];
      for (int32_t i = n - 1; i >= 0; i--) p[--cnt[c[p_new[i]]]] = p_new[i];
      c_new[p[0]] = 0;
      classes = 1;
      for (int32_t i = 1; i < n; i++) {
         pair<int32_t, int32_t> cur = {c[p[i]], c[(p[i] + (1 << h)) % n]};
         pair<int32_t, int32_t> prev = {c[p[i - 1]],
            c[(p[i - 1] + (1 << h)) % n]};
         if (cur != prev) ++classes;
         c_new[p[i]] = classes - 1;
      }
      c.swap(c_new);
   }
   return p;
}
vector<int32_t> suffix_array_construct(string s) {
   s += "$";
   // what about s += " "; ?
   vector<int32_t> sorted_shifts = sort_cyclic_shifts(s);
   // sorted_shifts.erase(sorted_shifts.begin()); - removes the element
   // corresponding to the empty suffix
   return sorted_shifts;
}

// burrow wheeler transform - find the string consisting of the last elements of the sorted rotated arrays

// inverse burrow wheeler transform

   string s;
   read_str(s);
   int n = s.size();
   vector<int> nextPosition;
   vector<vector<int>> positions(27);

   for (int i = 0; i < n; ++i)
      positions[max(0, s[i] - 'a' + 1)].push_back(i);

   for (int i = 0; i < 27; ++i)
      for (auto position : positions[i])
         nextPosition.push_back(position);

   int position = -1;
   for (int i = 0; i < n; ++i) {
      if (s[i] == '#') {
         position = i;
         break;
      }
   }

   assert(~position);

   for (int i = 1; i < n; ++i) {
      position = nextPosition[position];
      write_char(s[position]);
   }

   write_char('\n');
```

18 Template

```
#pragma GCC optimize("Ofast")
#pragma GCC target("avx")
#pragma GCC optimize("unroll-loops")

#include <bits/stdc++.h>

#include <ext/pb_ds/assoc_container.hpp>
```

```
#include <ext/pb_ds/tree_policy.hpp>
#include <ext/rope>

using namespace __gnu_pbds;
using namespace __gnu_cxx;
using namespace std;

#define int long long
#define double long double
#define Int signed
#define vi vector<int>
#define vI vector<Int>
#define vvi vector<vi>
#define vvI vector<vI>
#define vd vector<double>
#define vvd vector<vd>
#define pii pair<int, int>
#define vpii vector<pii>
#define vvpii vector<vpii>
#define mii map<int, int>
#define unordered_map gp_hash_table

#define OVERLOADED_MACRO(M, ...) _OVR(M, _COUNT_ARGS(__VA_ARGS__))(__VA_ARGS__)
#define _OVR(macroName, number_of_args) _OVR_EXPAND(macroName, number_of_args)
#define _OVR_EXPAND(macroName, number_of_args) macroName##number_of_args

#define _COUNT_ARGS(...) \
   _ARG_PATTERN_MATCH(__VA_ARGS__, 9, 8, 7, 6, 5, 4, 3, 2, 1)
#define _ARG_PATTERN_MATCH(_1, _2, _3, _4, _5, _6, _7, _8, _9, N, ...) N

#define rep(...) OVERLOADED_MACRO(rep, __VA_ARGS__)
#define repd(...) OVERLOADED_MACRO(repd, __VA_ARGS__)

#define rep3(i, a, b) for (int i = a; i < b; ++i)
#define rep2(i, n) rep3(i, 0, n)
#define repd3(i, a, b) for (int i = b - 1; i >= a; --i)
#define repd2(i, n) repd3(i, 0, n)
#define rep4(i, a, b, c) for (int i = a; i < b; i += c)

#define F first
#define S second

#define fastio \
   ios_base::sync_with_stdio(0); \
   cin.tie(0); \
   cout.tie(0);
#define pb push_back
#define mp make_pair
#define eb emplace_back
#define all(v) v.begin(), v.end()

#define bitcount __builtin_popcountll
// for trailing 1s, do trailing0(n + 1)
#define leading0 __builtin_clzll
#define trailing0 __builtin_ctzll
#define isodd(n) (n & 1)
#define iseven(n) (!(n & 1))

#define sz(v) ((int)v.size())
#define del_rep(v) \
   sort(all(v)); \
   v.erase(unique(all(v)), v.end());
#define checkbit(n, b) ((n >> b) & 1)

#ifdef DEBUG
#define debug(args...) \
   { \
      std::string _s = #args; \
      replace(_s.begin(), _s.end(), ',', ' '); \
      std::stringstream _ss(_s); \
      std::istream_iterator<std::string> _it(_ss); \
```

```
      err(_it, args); \
   }
#define print_container(v) \
   { \
      bool first = true; \
      os << "["; \
      for (auto x : v) { \
         if (!first) os << ", "; \
         os << x; \
         first = false; \
      } \
      return os << "]"; \
   }
void err(std::istream_iterator<std::string> it) {}
template <typename T, typename... Args>
void err(std::istream_iterator<std::string> it, T a, Args... args) {
   std::cerr << *it << " = " << a << std::endl;
   err(++it, args...);
}
template <typename T1, typename T2>
inline std::ostream& operator<<(std::ostream& os, const std::pair<T1, T2>& p) {
   return os << "(" << p.first << ", " << p.second << ")";
}
template <typename T>
inline std::ostream& operator<<(std::ostream& os, const std::vector<T>& v) {
   print_container(v);
}
template <typename T>
inline std::ostream& operator<<(std::ostream& os, const std::set<T>& v) {
   print_container(v);
}
template <typename T1, typename T2>
inline std::ostream& operator<<(std::ostream& os, const std::map<T1, T2>& v) {
   print_container(v);
}
template <typename T1, typename T2, class C>
inline std::ostream& operator<<(std::ostream& os,
                        const unordered_map<T1, T2, C>& v) {
   print_container(v);
}
template <typename T, class C>
inline std::ostream& operator<<(std::ostream& os,
                        const unordered_set<T, C>& v) {
   print_container(v);
}
template <typename T1, typename T2>
inline std::ostream& operator<<(std::ostream& os,
                        const std::multimap<T1, T2>& v) {
   print_container(v);
}
template <typename T>
inline std::ostream& operator<<(std::ostream& os, const std::multiset<T>& v) {
   print_container(v);
}
#else
#define debug(args...) 0
#endif

// order_of_key(k) - number of elements e such that func(e, k) returns true,
// where func is less or less_equal find_by_order(k) - kth element in the set
// counting from 0
//
typedef tree<int, null_type, less<int>, rb_tree_tag,
         tree_order_statistics_node_update>
   ordered_set;
typedef tree<int, null_type, less_equal<int>, rb_tree_tag,
         tree_order_statistics_node_update>
   ordered_multiset;

const int INF = 1e9;
const int LINF = INF * INF;
```

```cpp
const double EPS = 1e-9;
const double PI = acosl(-1);

// older machines
// use only for non-negative u, v
int32_t gcd(int32_t u, int32_t v) {
   int32_t shift;
   if (u == 0) return v;
   if (v == 0) return u;
   shift = __builtin_ctz(u | v);
   u >>= __builtin_ctz(u);
   do {
      v >>= __builtin_ctz(v);
      if (u > v) {
         swap(u, v);
      }
      v -= u;
   } while (v);
   return u << shift;
}

// use only for non-negative u, v
long long gcd(long long u, long long v) {
   int32_t shift;
   if (u == 0 || v == 0) return u + v;
   shift = __builtin_ctzll(u | v);
   u >>= __builtin_ctzll(u);
   do {
      v >>= __builtin_ctzll(v);
      if (u > v) {
         swap(u, v);
      }
      v -= u;
   } while (v);
   return u << shift;
}

struct custom_hash {
   // http://xorshift.di.unimi.it/splitmix64.c
   static uint64_t splitmix64(uint64_t x) {
      x += 0x9e3779b97f4a7c15;
      x = (x ^ (x >> 30)) * 0xbf58476d1ce4e5b9;
      x = (x ^ (x >> 27)) * 0x94d049bb133111eb;
      return x ^ (x >> 31);
   }
   size_t operator()(uint64_t x) const {
      static const uint64_t FIXED_RANDOM =
         chrono::steady_clock::now().time_since_epoch().count();
      return splitmix64(x + FIXED_RANDOM);
   }
   size_t operator()(pair<int, int> x) const {
      static const uint64_t FIXED_RANDOM =
         chrono::steady_clock::now().time_since_epoch().count();
      return splitmix64(FIXED_RANDOM + 31 * x.first + x.second);
   }
};

namespace IO {
const int BUFFER_SIZE = 1 << 15;
char input_buffer[BUFFER_SIZE];
int input_pos = 0, input_len = 0;
char output_buffer[BUFFER_SIZE];
int output_pos = 0;
char number_buffer[100];
uint8_t lookup[100];
void _update_input_buffer() {
   input_len = fread(input_buffer, sizeof(char), BUFFER_SIZE, stdin);
   input_pos = 0;
   if (input_len == 0) input_buffer[0] = EOF;
}
inline char next_char(bool advance = true) {
```

```
    if (input_pos >= input_len) _update_input_buffer();

    return input_buffer[advance ? input_pos++ : input_pos];
}

template <typename T>
inline void read_int(T& number) {
    bool negative = false;
    number = 0;

    while (!isdigit(next_char(false)))
        if (next_char() == '-') negative = true;

    do {
        number = 10 * number + (next_char() - '0');
    } while (isdigit(next_char(false)));

    if (negative) number = -number;
}

template <typename T, typename... Args>
inline void read_int(T& number, Args&... args) {
    read_int(number);
    read_int(args...);
}

void _flush_output() {
    fwrite(output_buffer, sizeof(char), output_pos, stdout);
    output_pos = 0;
}

inline void write_char(char c) {
    if (output_pos == BUFFER_SIZE) _flush_output();

    output_buffer[output_pos++] = c;
}

template <typename T>
inline void write_int(T number, char after = '\0') {
    if (number < 0) {
        write_char('-');
        number = -number;
    }
    int length = 0;
    while (number >= 10) {
        uint8_t lookup_value = lookup[number % 100];
        number /= 100;
        number_buffer[length++] = (lookup_value & 15) + '0';
        number_buffer[length++] = (lookup_value >> 4) + '0';
    }
    if (number != 0 || length == 0) write_char(number + '0');
    for (int i = length - 1; i >= 0; i--) write_char(number_buffer[i]);
    if (after) write_char(after);
}
void IOinit() {
    // Make sure _flush_output() is called at the end of the program.
    bool exit_success = atexit(_flush_output) == 0;
    assert(exit_success);
    for (int i = 0; i < 100; i++) lookup[i] = (i / 10 << 4) + i % 10;
}
} // namespace IO

using namespace IO;

template <class T, class F = multiplies<T>>
T power(T a, long long n, F op = multiplies<T>(), T e = {1}) {
    assert(n >= 0);
    T res = e;
    while (n) {
        if (n & 1) res = op(res, a);
        if (n >>= 1) a = op(a, a);
```

```cpp
    }
    return res;
}

template <unsigned Mod = 998'244'353>
struct Modular {
    using M = Modular;
    unsigned v;
    Modular(long long a = 0) : v((a %= Mod) < 0 ? a + Mod : a) {}
    M operator-() const { return M() -= *this; }
    M& operator+=(M r) {
        if ((v += r.v) >= Mod) v -= Mod;
        return *this;
    }
    M& operator-=(M r) {
        if ((v += Mod - r.v) >= Mod) v -= Mod;
        return *this;
    }
    M& operator*=(M r) {
        v = (uint64_t)v * r.v % Mod;
        return *this;
    }
    M& operator/=(M r) { return *this *= power(r, Mod - 2); }
    friend M operator+(M l, M r) { return l += r; }
    friend M operator-(M l, M r) { return l -= r; }
    friend M operator*(M l, M r) { return l *= r; }
    friend M operator/(M l, M r) { return l /= r; }
    friend bool operator==(M l, M r) { return l.v == r.v; }
    friend bool operator!=(M l, M r) { return l.v != r.v; }
    friend ostream& operator<<(ostream& os, M& a) { return os << a.v; }
    friend istream& operator>>(istream& is, M& a) {
        int64_t w;
        is >> w;
        a = M(w);
        return is;
    }
};

const int mod = 1e9 + 7;

using mint = Modular<>;

const int maxn = 5e5 + 5;
const int maxa = 1e6 + 5;
const int logmax = 25;

void solve(int case_no) {}

signed main() {
    IOinit();
    fastio;
    cout << setprecision(10) << fixed;
    int t = 1;
    // read_int(t);
    // cin >> t;
    for (int _t = 1; _t <= t; _t++) {
        // cout << "Case #" << _t << ": ";
        solve(_t);
    }
    _flush_output();
    return 0;
}
```