**Modular Multiplication:**

long long binMul(long long a, long long b, long long m) {

    a %= m;

    long long res = 0LL;

    while (b > 0) {

        if (b & 1) {

            res = (res + a) % m;

        }

        a = (a + a) % m;

        b >>= 1;

    }

    return res;

}

**Bin Pow:(binMul)**

long long binPow(long long a, long long b, long long m) {

    a %= m;

    long long res = 1;

    while (b > 0) {

        if (b & 1) {

            res = binMul(res, a, m);

        }

        a = binMul(a, a, m);

        b >>= 1;

    }

    return res;

}

**Modular Division:(binPow, binMul)**

long long binDiv(long long a, long long b, long long m) {

    int invOfB = binPow(b, m - 2, m);

    return binMul(a, invOfB, m);

}

**Inverse Mod:(binPow)**

long long inverseMod(long long n, long long mod) {

    return binPow(n, mod - 2, mod);

}

**Sieve of eratosthenes:**

void Seive(vector<int> &primes, vector<int> &f, int n) {

    f[1] = true;

    for (int i = 2; i \* i <= n; i++) {

        if (!f[i]) {

            for (int j = i \* i; j <= n; j += i) {

                f[j] = true;

            }

        }

    }

    for (int i = 2; i <= n; i++) {

        if (!f[i]) {

            primes.push\_back(i);

        }

    }

}

**Binary Exponential(With Mod):**

long long binMultiplication(long long a, long long b, long long m) {

    a %= m;

    long long res = 0LL;

    while(b > 0) {

        if(b & 1) {

            res = (res + a) % m;

        }

        a = (a + a) % m;

        b >>= 1;

    }

    return res;

}

long long binPow(long long a, long long b, long long m) {

    a %= m;

    long long res = 1;

    while (b > 0) {

        if (b & 1) {

            res = binMultiplication(res, a, m);

        }

        a = binMultiplication(a, a, m);

        b >>= 1;

    }

    return res;

}

long long binDivision(long long a, long long b, long long m) {

    int invOfB = binPow(b, m - 2, m);

    return binMultiplication(a, invOfB, m);

}

**Binary exponential:(Without Mod)**

long long binpow(long long a, long long b) {

    long long res = 1;

    while (b > 0) {

        if (b & 1)

            res = res \* a;

        a = a \* a;

        b >>= 1;

    }

    return res;

}

**GCD:**

int gcd(int a, int b) {

    if (b == 0) return a;

    return gcd(b, a % b);

}

**LCM:(gcd)**

long long lcm(long long a, long long b) {

    return (a / gcd(a, b) \* b);

}

**Extended Euclidean Algorithm:**

int extendedGcd(int a, int b, int &x, int &y) {

    if (b == 0) {

        x = 1;

        y = 0;

        return a;

    }

    int x1, y1;

    int d = extendedGcd(b, a % b, x1, y1);

    x = y1;

    y = x1 - y1 \* (a / b);

    return d;

}

**nCr:**

long long nCr(long long a, long long b) {

    long long x = max(a - b, b), ans = 1;

    for (long long K = a, L = 1; K >= x + 1; K--, L++) {

        ans = ans \* K;

        ans /= L;

    }

    return ans;

}

**nCrMod:{factorial pre computation needed}(inverseMod)**

long long fact[200005];

long long nCrMod(long long n,long long r, long long mod) {

    return (((fact[n]\*inverseMod(fact[r], mod))%mod)\*

inverseMod(fact[n-r], mod))%mod;

}

**nCrMod:{factorial pre-computeded}(inverseMod)**

long long fact[200005];

void factorial(long long n, long long mod){

    for (long long i = 1; i <= n; i++) {

        fact[i] = (fact[i - 1] \* i) % mod;

    }

}

long long nCrMod(long long n, long long r, long long mod) {

    return (((fact[n] \* inverseMod(fact[r], mod)) % mod) \*

            inverseMod(fact[n - r], mod)) % mod;

}

**nCr Mod:{Using struct}(inverseMod)**

struct nCrMod {

// declare a object [nCrMod nc(n, mod)] like this

    long long N;

    vector<long long> fact, inv;

    nCrMod(long long n, long long mod) {

        N = n;

        fact.assign(n + 5, 0);

        inv.assign(n + 5, 0);

        fact[0] = 1;

        for (long long i = 1; i <= N; i++) {

            fact[i] = (fact[i - 1] \* i) % mod;

        }

        inv[N] = inverseMod(fact[N], mod);

        for (long long i = N - 1; i >= 0; i--) {

            inv[i] = (inv[i + 1] \* (i + 1)) % mod;

        }

    }

    long long find(long long n, long long r, long long mod) {

        if (n < r || r < 0) return 0;

        long long temp = (inv[r] \* inv[n - r]) % mod;

        return (temp \* fact[n]) % mod;

    }

};

**nCrMod:(No pre-requisite)**

struct nCrMod {

    // declare a object [nCrMod nc(n, mod)] like this

    long long N;

    vector<long long> fact, inv;

    long long binMul(long long a, long long b, long long m) {

        a %= m;

        long long res = 0LL;

        while (b > 0) {

            if (b & 1) {

                res = (res + a) % m;

            }

            a = (a + a) % m;

            b >>= 1;

        }

        return res;

    }

    long long binPow(long long a, long long b, long long m) {

        a %= m;

        long long res = 1;

        while (b > 0) {

            if (b & 1) {

                res = binMul(res, a, m);

            }

            a = binMul(a, a, m);

            b >>= 1;

        }

        return res;

    }

    long long inverseMod(long long n, long long mod) {

        return binPow(n, mod - 2, mod);

    }

    nCrMod(long long n, long long mod) {

        N = n;

        fact.assign(n + 5, 0);

        inv.assign(n + 5, 0);

        fact[0] = 1;

        for (long long i = 1; i <= N; i++) {

            fact[i] = (fact[i - 1] \* i) % mod;

        }

        inv[N] = inverseMod(fact[N], mod);

        for (long long i = N - 1; i >= 0; i--) {

            inv[i] = (inv[i + 1] \* (i + 1)) % mod;

        }

    }

    long long find(long long n, long long r, long long mod) { //use this

        if (n < r || r < 0) return 0;

        long long temp = (inv[r] \* inv[n - r]) % mod;

        return (temp \* fact[n]) % mod;

    }

};