UFRJ - Turma do fundão

Contents

1 Number Theory

1.1 Sieve of Eratosthenes

```
//O(n)
int lp[MAXN], pr[MAXN];
int cnt;

void sieve( int n ) {
  for( int i = 2 ; i <= n ; ++i ) {
    if( lp[i] == 0 ) lp[i] = pr[cnt++] = i;
    for( int j = 0 ; j < cnt && pr[j]<=lp[i] && i * pr[j] <= n ; ++j )
      lp[i * pr[j]] = pr[j];
  }
}

// O(n log log n)
int n;
vector<char> is_prime(n+1, true);
is_prime[0] = is_prime[1] = false;
for (int i = 2; i * i <= n; i++) {
    if (is_prime[i]) {
        for (int j = i * i; j <= n; j += i)
            is_prime[j] = false;
    }
}
```

1.2 Discrete logarithm

```
// find k such that a^k = m mod(p), with p prime
// O(sqrt(n))
ll bb( ll a, ll m, ll p ) {
  unordered_map<ll, ll> mp;
  ll b = 1, an = a;
  while( b * b < p ) b++, an = ( an * a ) % p;
  ll bs = m;
  for( ll i = 0 ; i <= b ; ++i ) {
    mp[bs] = i;
    bs = ( bs * a ) % p;
  }
  ll gs = an;
  for( ll i = 1 ; i <= b ; ++i ) {
    if( mp.count( gs ) ) return ( b * i - mp[gs] );
    gs = ( gs * an ) % p;
  }
  return -1;
}

// bellow works for some C composite A^k = B mod C
// O(sqrt(n)), do not forget fastexp
#define ll long long
ll bb(ll A, ll B, ll C) {
  A %= C, B %= C;
  if(B == 1) return 0;
  ll k = 0;
  ll tmp = 1;
  for(int d = __gcd(A, C) ; d != 1 ; d = __gcd(A, C)) {
    if(B%d) return -1;
    B /= d, C /= d;
    tmp = tmp*(A/d)%C;
    ++k;
    if(tmp == B) return k;
  }
  unordered_map<ll, int> mp;
  ll mul = B;
  ll m = sqrt(C);
  for(ll j = 0 ; j < m ; ++j)
    mp[mul] = j, mul = mul*A%C;
  ll am = fastexp(A, m, C);
  mul = tmp;
  for(ll j = 1 ; j <= m + 1 ; ++j) {
    mul = mul*am%C;
    if(mp.count(mul)) return j*m-mp[mul]+k;
  }
  return -1;
}
```

1.3 GCD/LCM/Fast expo/Mul mod

```
#define ll long long
//O(log n)
ll gcd( ll a, ll b ) {
  return b ? gcd( b, a % b ) : a;
}
//O(log n)
ll lcm( ll a, ll b ) {
  return a * ( b / gcd( a, b ) );
}
//O(log n)
ll mulmod( ll a, ll b, ll m ) {
  ll r = 0 ;
  for( a %= m ; b ; b >>= 1, a = ( a * 2 ) % m )
    if( b & 1 ) r = ( r + a ) % m;
  return r;
}

//O(1)?
typedef long double ld;
ll mulmod( ll a, ll b, ll m ) {
  ll q = (ld) a * (ld) b / (ld) m;
  ll r = a * b - q * m;
  return ( r + m ) % m;
}

// a^b mod m | O(log b)
ll fastexp( ll a, ll b, ll m ) {
  ll r = 1;
  for( a %= m ; b ; b >>= 1, a = mulmod( a, a, m ) )
    if( b & 1 ) r = mulmod( r, a, m );
  return r;
}
```

```
// Multiplicative Inverse
ll inv[MAXN];
inv[1] = 1;
for( int i = 2 ; i < MOD ; ++i )
  inv[i] = (MOD - (MOD/i)*inv[MOD%i]%MOD)%MOD;

//O(sqrt(n))
vector<int> allDivisors( int n ) {
  vector<int> f;
  for( int i = 1 ; i <= (int)sqrt( n ) ; ++i ) {
    if( n % i == 0 ) {
      if( n / i == i ) f.push_back( i );
      else f.push_back( i ), f.push_back( n / i );
    }
  }
  return f;
}

// Recurrence using matriz
// h[i+2] = a1*h[i+1] + a0*h[i]
// [ h[i] h[i-1] ] = [ h[1] h[0] ] * [ a1 1 ] ^ (i - 1)[ a0 0 ]

// Fibonacci in logarithm time
// f(2*k) = f(k)*(2*f(k + 1) - f(k))
// f(2*k + 1) = f(k)^2 + f(k + 1)^2

// Catalan
// Cn = (2n)! / ((n+1)! * n!)
// 1, 1, 2, 5, 14, 42, 132, 429, 1430, 4862, 16796, 58786, 208012, 742900,
    2674440
cat[n] = (2*(2*n-1)/(n+1)) * cat[n-1]

// Stirling
// S(n, 1) = S(n, n) = 1
// S(n, k) = k*S(n-1, k) + S(n-1, k-1)

// Burnside's Lemma
// Counts the number of equivalence classes in a set.
// Let G be a group of operations acting on a set X. The number of equivalence
    classes given those operations |X/G| satisfies:
//
// |X/G| = 1/|G| * sum(I(g)) for each g in G
//
// Being I(g) the number of fixed points given the operation g.
```

1.4 Euclidian + Chinese Reminder

```
#define ll long long
// Solve: x * a + y * b = gcd(a,b) | O(log n)
void euclid( ll a, ll b, ll &x, ll &y, ll &gcd ) {
  if( b ) euclid( b, a % b, y, x, gcd ), y -= x * ( a / b );
  else x = 1, y = 0, gcd = a;
}

// Chinese remainder, solves t = a mod m1 ; t = b mod m2 ; ans = t mod lcm( m1,
    m2 )
// O(log n)
bool chinese( ll a, ll b, ll m1, ll m2, ll &ans, ll &lcm ) {
  ll x, y, g, c = b - a;
  euclid( m1, m2, x, y, g );
  if( c % g ) return false;

  lcm = m1 / g * m2;
  ans = ( ( a + c / g * x % ( m2 / g ) * m1 ) % lcm + lcm ) % lcm;
  return true;
}
```

```
// Solve: a * x + b * y = c | O(log n)
bool euclidFind( ll a, ll b, ll c, ll &x0, ll &y0, ll &g ) {
  euclid( abs( a ), abs( b ), x0, y0, g );
  if( c % g ) return false;
  x0 *= c / g, y0 *= c / g;
  if( a < 0 ) x0 = -x0;
  if( b < 0 ) y0 = -y0;
  return true;
}

void shift( ll &x, ll &y, ll a, ll b, ll cnt ) {
  x += cnt * b;
  y -= cnt * a;
}

// Count all solutions in range | O(solutions * log n)
// it can be very slow
ll all( ll a, ll b, ll c, ll minx, ll maxx, ll miny, ll maxy ) {
  ll x, y, g;
  if( !find_any_solution( a, b, c, x, y, g ) ) return 0;
  a /= g, b /= g;
  ll sign_a = a > 0 ? +1 : -1;
  ll sign_b = b > 0 ? +1 : -1;
  shift( x, y, a, b, ( minx - x ) / b );
  if( x < minx ) shift( x, y, a, b, sign_b );
  if( x > maxx ) return 0;
  ll lx1 = x;
  shift( x, y, a, b, ( maxx - x ) / b );
  if( x > maxx ) shift( x, y, a, b, -sign_b );
  ll rx1 = x;
  shift( x, y, a, b, - ( miny - y ) / a );
  if( y < miny ) shift( x, y, a, b, -sign_a );
  if( y > maxy ) return 0;
  ll lx2 = x;
  shift( x, y, a, b, - ( maxy - y ) / a );
  if( y > maxy ) shift( x, y, a, b, sign_a );
  ll rx2 = x;
  if( lx2 > rx2 ) swap( lx2, rx2 );
  ll lx = max( lx1, lx2 );
  ll rx = min( rx1, rx2 );
  if( lx > rx ) return 0;
  return ( rx - lx ) / abs( b ) + 1;
}
```

1.5 Primitive root

```
// do not forget fastexp
// some numbers that have primitive root:
// 1, 2, 3, 4, 5, 6, 7, 9, 10, 11, 13, 14, 17, 18, 19, 22, 23, 25, 26, 27, 29
// O(n) eu acho
#define ll long long

ll root( ll p ) {
  ll n = p-1;
  vector<ll> fact;
  for( int i = 2 ; i * i <= n ; ++i ) if( n % i == 0 ) {
    fact.push_back( i );
    while( n % i == 0 ) n /= i;
  }
  if( n > 1 ) fact.push_back( n );
  for( int res = 2 ; res <= p ; ++res ) {
    bool ok = true;
    for( size_t i = 0 ; i < fact.size() && ok ; ++i )
      ok &= fastexp( res, ( p - 1 ) / fact[i], p ) != 1;
    if( ok )  return res;
  }
```

```
  return -1;
}
```

1.6 Miller rabin

```
// Miller-Rabin - Primarily Test O(k*log^3(n))
#define ll long long
bool miller( ll a, ll n ) {
  if( a >= n ) return 1;
  ll s = 0, d = n-1;
  while( d & 1 == 0 and d ) d >>= 1, ++s;
  ll x = fastexp( a, d, n );
  if( x == 1 or x == n - 1 ) return 1;
  for( int r = 0 ; r < s ; ++r, x = mulmod( x, x, n ) ) {
    if( x == 1 ) return 0;
    if( x == n - 1 ) return 1;
  }
  return 0;
}

bool isprime( ll n ) {
  int base[] = {2, 3, 5, 7, 11, 13, 17, 19, 23, 29, 31, 37};
  for( int i = 0 ; i < 12 ; ++i ) if( !miller( base[i], n ) ) return 0;
  return 1;
}
```

1.7 Prime factors

```
// Prime factors (up to 9*10^13. For greater see Pollard Rho) | O(sqrt( n ))
// sieve( sqrt( n ) ); to get all primes until sqrt(n)
vector<int> factors;
int ind=0, pf = pr[0];
while( pf * pf <= n ) {
  while( n%pf == 0 ) n /= pf, factors.push_back( pf );
  pf = pr[++ind];
}
if( n != 1 ) factors.push_back( n );

vector<ll> divisors( ll n) {
  vector<ll> v;
  for( ll i = 1 ; i <= sqrt( n ) ; ++i ){
    if( n % i == 0 ) {
      if( n / i == i ) v.push_back( i );
      else v.push_back( i ), v.push_back( n/i );
    }
  }
  return v;
}
```

1.8 Pollard Rho

```
// Pollard Rho - Integer factoring O(n^1/4)
// Do not forget mulmod, gcd, miller-rabin
#define ll long long
#define ull unsigned ll
#define pb push_back

std::mt19937 rng( ( int ) std::chrono::steady_clock::now().time_since_epoch().
     count() );
ull func( ull x, ull n, ull c ) { return ( mulmod( x, x, n ) + c ) % n; }

ull pollard( ull n ) {
  ull x, y, d, c;
  ull pot, lam;
  if( n & 1 == 0 ) return 2;
  if( isprime( n ) ) return n;
  while( 1 ) {
    y = x = 2; d = 1;
    pot = lam = 1;
    while( 1 ) {
      c = rng() % n;
      if( c != 0 && ( c + 2 ) % n != 0 ) break;
    }
    while( 1 ) {
      if( pot == lam ) x = y, pot <<= 1, lam = 0;
      y = func( y, n, c );
      ++lam;
      d = gcd( x >= y ? x - y : y - x, n );
      if( d > 1 ) {
        if( d == n ) break;
        else return d;
      }
    }
  }
}

void fator( ll n, vector<ll>& v ) {
  if( isprime( n ) ) { v.pb(n); return; }
  ll f = pollard( n );
  fator( f, v ); fator( n / f, v );
}

void fator( ull n, vector<ull> &v ) {
  if( isprime( n ) ) { v.pb( n ); return; }
  vector<ull> w, t; w.pb( n ); t.pb( 1 );

  while( !w.empty() ) {
    ull bck = w.back();
    ull div = pollard( bck );
    if( div == w.back() ) {
      int amt = 0;
      for( int i = 0 ; i < ( int ) w.size() ; ++i ) {
        int cur = 0;
        while( w[i] % div == 0 ) w[i] /= div, ++cur;
        amt += cur * t[i];
        if( w[i] == 1 ) {
          swap(w[i], w.back());
          swap(t[i], t.back());
          w.pop_back();
          t.pop_back();
        }
      }
      while( amt-- ) v.pb( div );
    } else {
      int amt = 0;
      while( w.back() % div == 0 ) {
        w.back() /= div;
        ++amt;
      }
      amt *= t.back();
      if( w.back() == 1 ) {
        w.pop_back();
        t.pop_back();
      }

      w.pb( div );
      t.pb( amt );
    }
  }
  sort( v.begin(), v.end() );
}
```

1.9 ϕ of Euler

```
// numeros coprimos menores ou iguais a n
// O(sqrt(n))
int phi(int n) {
  int result = n;
  for( int i = 2 ; i * i <= n ; ++i ){
    if( n % i == 0 ){
      while( n % i == 0 ) n /= i;
      result -= result / i;
    }
  }
  if( n > 1 ) result -= result / n;
    return result;
}
// Compute array with all phi until N
// O(n*?) it is not so slow, check if its better to
// O(k*sqrt(n)) or this | this one was faster on SPOJ
int phi[MAXN];
void totient( int N ) {
  for( int i = 1 ; i < N ; ++i)  phi[i]=i;
  for( int i = 2 ; i < N ; i += 2 ) phi[i] >>= 1;
  for( int j = 3 ; j < N ; j += 2 ) if( phi[j]==j ) {
    --phi[j];
    for( int i = 2 * j ; i < N ; i += j ) phi[i] = phi[i] / j * ( j - 1 );
  }
}
```

1.10 Compute prime factors

```
// Find all prime factors | O(n^1/3) ?
// here we find the smallest finite base of a fraction a/b
#define ll long long
int main() {
    scanf("%lld %lld", &a, &b);

    ll g = __gcd(a, b);
    b /= g;

    cur = b;
    for(ll i = 2; i <= cbrt(cur); i++) {
        if(cur % i == 0) {
            ans *= i;
            while(cur % i == 0) cur /= i;
        }
    }

    ll sq = round(sqrt(cur));
    if(sq * sq == cur) cur = sq;

    printf("%lld\n", max(2LL, ans * cur));
    return 0;
}
```

1.11 Finite Field operations

```
// Operations with mod p :)
typedef long long LL;

template<int p> struct FF {
  LL val;

  FF(const LL x=0) { val = (x % p + p) % p; }

  bool operator==(const FF<p>& other) const { return val == other.val; }
  bool operator!=(const FF<p>& other) const { return val != other.val; }

  FF operator+() const { return val; }
  FF operator-() const { return -val; }

  FF& operator+=(const FF<p>& other) { val = (val + other.val) % p; return *this
      ; }
  FF& operator-=(const FF<p>& other) { *this += -other; return *this; }
  FF& operator*=(const FF<p>& other) { val = (val * other.val) % p; return *this
      ; }
  FF& operator/=(const FF<p>& other) { *this *= other.inv(); return *this; }

  FF operator+(const FF<p>& other) const { return FF(*this) += other; }
  FF operator-(const FF<p>& other) const { return FF(*this) -= other; }
  FF operator*(const FF<p>& other) const { return FF(*this) *= other; }
  FF operator/(const FF<p>& other) const { return FF(*this) /= other; }

  static FF<p> pow(const FF<p>& a, LL b) {
    if (!b) return 1;
    return pow(a * a, b >> 1) * (b & 1 ? a : 1);
  }

  FF<p> pow(const LL b) const { return pow(*this, b); }
  FF<p> inv() const { return pow(p - 2); }
};

template<int p> FF<p> operator+(const LL lhs, const FF<p>& rhs) { return FF<p>(
    lhs) += rhs; }
template<int p> FF<p> operator-(const LL lhs, const FF<p>& rhs) { return FF<p>(
    lhs) -= rhs; }
template<int p> FF<p> operator*(const LL lhs, const FF<p>& rhs) { return FF<p>(
    lhs) *= rhs; }
template<int p> FF<p> operator/(const LL lhs, const FF<p>& rhs) { return FF<p>(
    lhs) /= rhs; }

typedef FF<1000000007> num;
```

2 Numeric

2.1 Binomial

```
// compute binomial coeficient O(n*k)
inv[(n-2)!]=inv[(n-1)!] *(n-1)
fat[1]=1, inv[0]=1;
for(int i=2;i<=n;i++){
  fat[i]=(fat[i-1]*i)%mod;
}
inv[n-1]=power(fat[n-1], mod-2, mod);
for(int i=n-2;i>=1;i--){
  inv[i]=(inv[i+1]*(i+1))%mod;
}
for(int i=1;i<=n;i++){
  esc[i][i]=1ll;
  esc[i][0]=1ll;
  for(int j=1;j<=i-1;j++){
    esc[i][j]=((fat[i]*inv[j])%mod*inv[i-j])%mod;
  }
}
```

2.2 Simpson Rule

```
// Numerical integration O(n)

double f( double x ) {

}

double simpson( double a, double b, int n = 1e6 ) {
  double h = ( b - a ) / n;
  double s = f( a ) + f( b );
  for( int i = 1 ; i < n ; i += 2 ) s += 4 * f( a + h * i );
  for( int i = 2 ; i < n ; i += 2 ) s += 2 * f( a + h * i );
  return s * h / 3;
}
```

2.3 Runge-kutta ODE

```
// solve ODE O(n)
#define EPS 1e-5
double runge_kutta(double (*f)(), double t, double tend, double x) {
  for( double h = EPS; t < tend; ) {
    if( t + h >= tend ) h = tend - t;
    double k1 = h * f( t, x );
    double k2 = h * f( t + h/2, x + k1/2 );
    double k3 = h * f( t + h/2, x + k2/2 );
    double k4 = h * f( t + h  , x + k3);
    x += (k1 + 2 * k2 + 2 * k3 + k4) / 6;
    t += h;
  }
  return x;
}
```

2.4 Fast Fourier transform

```
// fast multiply, O(n*log(n))
namespace fft {
  typedef double dbl;

  struct num {
    dbl x, y;
    num() { x = y = 0; }
    num(dbl x, dbl y) : x(x), y(y) {}
  };

  inline num operator+ (num a, num b) { return num(a.x + b.x, a.y + b.y); }
  inline num operator- (num a, num b) { return num(a.x - b.x, a.y - b.y); }
  inline num operator* (num a, num b) { return num(a.x * b.x - a.y * b.y, a.x *
      b.y + a.y * b.x); }
  inline num conj(num a) { return num(a.x, -a.y); }

  int base = 1;
  vector<num> roots = {{0, 0}, {1, 0}};
  vector<int> rev = {0, 1};

  const dbl PI = acosl(-1.0);

  void ensure_base(int nbase) {
    if(nbase <= base) return;

    rev.resize(1 << nbase);
    for(int i=0; i < (1 << nbase); i++) {
      rev[i] = (rev[i >> 1] >> 1) + ((i & 1) << (nbase - 1));
    }
    roots.resize(1 << nbase);

    while(base < nbase) {
      dbl angle = 2*PI / (1 << (base + 1));
      for(int i = 1 << (base - 1); i < (1 << base); i++) {
        roots[i << 1] = roots[i];
        dbl angle_i = angle * (2 * i + 1 - (1 << base));
        roots[(i << 1) + 1] = num(cos(angle_i), sin(angle_i));
      }
      base++;
    }
  }

  void fft(vector<num> &a, int n = -1) {
    if(n == -1) {
      n = a.size();
    }
    assert((n & (n-1)) == 0);
    int zeros = __builtin_ctz(n);
    ensure_base(zeros);
    int shift = base - zeros;
    for(int i = 0; i < n; i++) {
      if(i < (rev[i] >> shift)) {
        swap(a[i], a[rev[i] >> shift]);
      }
    }
    for(int k = 1; k < n; k <<= 1) {
      for(int i = 0; i < n; i += 2 * k) {
        for(int j = 0; j < k; j++) {
          num z = a[i+j+k] * roots[j+k];
          a[i+j+k] = a[i+j] - z;
          a[i+j] = a[i+j] + z;
        }
      }
    }
  }

  vector<num> fa, fb;
  vector<int> multiply(vector<int> &a, vector<int> &b) {
    int need = a.size() + b.size() - 1;
    int nbase = 0;
    while((1 << nbase) < need) nbase++;
    ensure_base(nbase);
    int sz = 1 << nbase;
    if(sz > (int) fa.size()) {
      fa.resize(sz);
    }
    for(int i = 0; i < sz; i++) {
      int x = (i < (int) a.size() ? a[i] : 0);
      int y = (i < (int) b.size() ? b[i] : 0);
      fa[i] = num(x, y);
    }
    fft(fa, sz);
    num r(0, -0.25 / sz);
    for(int i = 0; i <= (sz >> 1); i++) {
      int j = (sz - i) & (sz - 1);
      num z = (fa[j] * fa[j] - conj(fa[i] * fa[i])) * r;
      if(i != j) {
        fa[j] = (fa[i] * fa[i] - conj(fa[j] * fa[j])) * r;
      }
      fa[i] = z;
    }
    fft(fa, sz);
    vector<int> res(need);
    for(int i = 0; i < need; i++) {
      res[i] = fa[i].x + 0.5;
    }
```

```
    return res;
  }

  vector<int> multiply_mod(vector<int> &a, vector<int> &b, int m, int eq = 0) {
    int need = a.size() + b.size() - 1;
    int nbase = 0;
    while ((1 << nbase) < need) nbase++;
    ensure_base(nbase);
    int sz = 1 << nbase;
    if (sz > (int) fa.size()) {
      fa.resize(sz);
    }
    for (int i = 0; i < (int) a.size(); i++) {
      int x = (a[i] % m + m) % m;
      fa[i] = num(x & ((1 << 15) - 1), x >> 15);
    }
    fill(fa.begin() + a.size(), fa.begin() + sz, num {0, 0});
    fft(fa, sz);
    if (sz > (int) fb.size()) {
      fb.resize(sz);
    }
    if (eq) {
      copy(fa.begin(), fa.begin() + sz, fb.begin());
    } else {
      for (int i = 0; i < (int) b.size(); i++) {
        int x = (b[i] % m + m) % m;
        fb[i] = num(x & ((1 << 15) - 1), x >> 15);
      }
      fill(fb.begin() + b.size(), fb.begin() + sz, num {0, 0});
      fft(fb, sz);
    }
    dbl ratio = 0.25 / sz;
    num r2(0, -1);
    num r3(ratio, 0);
    num r4(0, -ratio);
    num r5(0, 1);
    for (int i = 0; i <= (sz >> 1); i++) {
      int j = (sz - i) & (sz - 1);
      num a1 = (fa[i] + conj(fa[j]));
      num a2 = (fa[i] - conj(fa[j])) * r2;
      num b1 = (fb[i] + conj(fb[j])) * r3;
      num b2 = (fb[i] - conj(fb[j])) * r4;
      if (i != j) {
        num c1 = (fa[j] + conj(fa[i]));
        num c2 = (fa[j] - conj(fa[i])) * r2;
        num d1 = (fb[j] + conj(fb[i])) * r3;
        num d2 = (fb[j] - conj(fb[i])) * r4;
        fa[i] = c1 * d1 + c2 * d2 * r5;
        fb[i] = c1 * d2 + c2 * d1;
      }
      fa[j] = a1 * b1 + a2 * b2 * r5;
      fb[j] = a1 * b2 + a2 * b1;
    }
    fft(fa, sz);
    fft(fb, sz);
    vector<int> res(need);
    for (int i = 0; i < need; i++) {
      long long aa = fa[i].x + 0.5;
      long long bb = fb[i].x + 0.5;
      long long cc = fa[i].y + 0.5;
      res[i] = (aa + ((bb % m) << 15) + ((cc % m) << 30)) % m;
    }
    return res;
  }

  vector<int> square_mod(vector<int> &a, int m) {
    return multiply_mod(a, a, m, 1);
  }
}
```

2.5 Simplex method for LP

```
// maximize     c^T x
// subject to   Ax <= b
//              x >= 0
//
// A -- an m x n matrix
// b -- an m-dimensional vector
// c -- an n-dimensional vector
// x -- a vector where the optimal solution will be stored
// O(n^3 * error) | as the epsilon decrease, error increase
typedef long double DOUBLE;
typedef vector<DOUBLE> VD;
typedef vector<VD> VVD;
typedef vector<int> VI;
const DOUBLE EPS = 1e-9;

struct LPSolver {
  int m, n;
  VI B, N;
  VVD D;

  LPSolver(const VVD &A, const VD &b, const VD &c) :
    m(b.size()), n(c.size()), N(n + 1), B(m), D(m + 2, VD(n + 2)) {
    for (int i = 0; i < m; i++) for (int j = 0; j < n; j++) D[i][j] = A[i][j];
    for (int i = 0; i < m; i++) { B[i] = n + i; D[i][n] = -1; D[i][n + 1] = b[i
        ]; }
    for (int j = 0; j < n; j++) { N[j] = j; D[m][j] = -c[j]; }
    N[n] = -1; D[m + 1][n] = 1;
  }

  void Pivot(int r, int s) {
    for (int i = 0; i < m + 2; i++) if (i != r)
      for (int j = 0; j < n + 2; j++) if (j != s)
        D[i][j] -= D[r][j] * D[i][s] / D[r][s];
    for (int j = 0; j < n + 2; j++) if (j != s) D[r][j] /= D[r][s];
    for (int i = 0; i < m + 2; i++) if (i != r) D[i][s] /= -D[r][s];
    D[r][s] = 1.0 / D[r][s];
    swap(B[r], N[s]);
  }

  bool Simplex(int phase) {
    int x = phase == 1 ? m + 1 : m;
    while (true) {
      int s = -1;
      for (int j = 0; j <= n; j++) {
        if (phase == 2 && N[j] == -1) continue;
        if (s == -1 || D[x][j] < D[x][s] || D[x][j] == D[x][s] && N[j] < N[s]) s
            = j;
      }
      if (D[x][s] > -EPS) return true;
      int r = -1;
      for (int i = 0; i < m; i++) {
        if (D[i][s] < EPS) continue;
        if (r == -1 || D[i][n + 1] / D[i][s] < D[r][n + 1] / D[r][s] ||
          (D[i][n + 1] / D[i][s]) == (D[r][n + 1] / D[r][s]) && B[i] < B[r]) r =
            i;
      }
      if (r == -1) return false;
      Pivot(r, s);
    }
  }

  DOUBLE Solve(VD &x) {
    int r = 0;
    for (int i = 1; i < m; i++) if (D[i][n + 1] < D[r][n + 1]) r = i;
    if (D[r][n + 1] < -EPS) {
```

```
    Pivot(r, n);
    if (!Simplex(1) || D[m + 1][n + 1] < -EPS) return -numeric_limits<DOUBLE
        >::infinity();
    for (int i = 0; i < m; i++) if (B[i] == -1) {
      int s = -1;
      for (int j = 0; j <= n; j++)
        if (s == -1 || D[i][j] < D[i][s] || D[i][j] == D[i][s] && N[j] < N[s])
            s = j;
      Pivot(i, s);
    }
  }
  if (!Simplex(2)) return numeric_limits<DOUBLE>::infinity();
  x = VD(n);
  for (int i = 0; i < m; i++) if (B[i] < n) x[B[i]] = D[i][n + 1];
  return D[m][n + 1];
 }
};
```

2.6 Gaussian elimination

```
// O(n^3)
// return determinant
// a will be inverted
// b will return x
const double EPS = 1e-10;

double Gauss( vector<vector<double> > &a, vector<vector<double> > &b ) {
  const int n = a.size();
  const int m = b[0].size();
  vector<int> irow( n ), icol( n ), ipiv( n );
  double det = 1;

  for( int i = 0 ; i < n ; ++i ) {
    int pj = -1, pk = -1;
    for( int j = 0 ; j < n ; ++j ) if( !ipiv[j] )
      for( int k = 0 ; k < n ; ++k ) if( !ipiv[k] )
        if( pj == -1 || fabs( a[j][k] ) > fabs( a[pj][pk] ) ) { pj = j; pk = k;
            }
    if( fabs( a[pj][pk] ) < EPS ) { /* Error matrix is singular. */ }
    ++ipiv[pk];
    swap( a[pj], a[pk] );
    swap( b[pj], b[pk] );
    if( pj != pk ) det *= -1;
    irow[i] = pj;
    icol[i] = pk;
    double c = 1.0 / a[pk][pk];
    det *= a[pk][pk];
    a[pk][pk] = 1.0;
    for( int p = 0 ; p < n ; ++p ) a[pk][p] *= c;
    for( int p = 0 ; p < m ; ++p ) b[pk][p] *= c;
    for( int p = 0 ; p < n ; ++p ) if( p != pk ) {
      c = a[p][pk];
      a[p][pk] = 0;
      for( int q = 0 ; q < n ; ++q ) a[p][q] -= a[pk][q] * c;
      for( int q = 0 ; q < m ; ++q ) b[p][q] -= b[pk][q] * c;
    }
  }

  for( int p = n - 1 ; p >= 0 ; --p ) if( irow[p] != icol[p] )
    for( int k = 0 ; k < n ; ++k ) swap( a[k][irow[p]], a[k][icol[p]] );
  return det;
}
```

2.7 Karatsuba

```
//O(n^1.6) All sizes MUST BE power of two
#define MAX 262144
#define MOD 1000000007

unsigned long long temp[128];
int ptr = 0, buffer[MAX * 6];
// the result is stored in *a
void karatsuba(int n, int *a, int *b, int *res){
  int i, j, h;
  if (n < 17){
    for (i = 0; i < (n + n); i++) temp[i] = 0;
    for (i = 0; i < n; i++){
      if (a[i]){
        for (j = 0; j < n; j++){
          temp[i + j] += ((long long)a[i] * b[j]);
        }
      }
    }
    for (i = 0; i < (n + n); i++) res[i] = temp[i] % MOD;
    return;
  }

  h = n >> 1;
  karatsuba(h, a, b, res);
  karatsuba(h, a + h, b + h, res + n);
  int *x = buffer + ptr, *y = buffer + ptr + h, *z = buffer + ptr + h + h;

  ptr += (h + h + n);
  for (i = 0; i < h; i++){
    x[i] = a[i] + a[i + h], y[i] = b[i] + b[i + h];
    if (x[i] >= MOD) x[i] -= MOD;
    if (y[i] >= MOD) y[i] -= MOD;
  }

  karatsuba(h, x, y, z);
  for (i = 0; i < n; i++) z[i] -= (res[i] + res[i + n]);
  for (i = 0; i < n; i++){
    res[i + h] = (res[i + h] + z[i]) % MOD;
    if (res[i + h] < 0) res[i + h] += MOD;
  }
  ptr -= (h + h + n);
}

int mul(int n, int *a, int m, int *b){
  int i, r, c = (n < m ? n : m), d = (n > m ? n : m), *res = buffer + ptr;
  r = 1 << (32 - __builtin_clz(d) - (__builtin_popcount(d) == 1));
  for (i = d; i < r; i++) a[i] = b[i] = 0;
  for (i = c; i < d && n < m; i++) a[i] = 0;
  for (i = c; i < d && m < n; i++) b[i] = 0;

  ptr += (r << 1), karatsuba(r, a, b, res), ptr -= (r << 1);
  for (i = 0; i < (r << 1); i++) a[i] = res[i];
  return (n + m - 1);
}
```

2.8 Inclusion-Exclusion principle

```
// inclusion exclusion principle
int n, k, res;
vector<int>pr;

void solve(int a, int p, ll x){
  if( x > n ) return;
  if( p == -1 ){
    if( x == 1 ) return;
    res += ( a%2 == 1 ? -1 : 1 ) * n / x;
    return;
```

```
    }
    solve( a, p - 1, x );
    solve( a + 1, p - 1, x * pr[p] );
}
```

3 Graph algorithms

3.1 Dijkstra Shortest path

```
// Shortest path from start to any other vertex O( (V + E) * log(E) )
// Doesnt work with negative weights (use SPFA)
#define ll long long
#define INF 0x3f3f3f3f3f3f3f3f
vector<ll> dk( int start, int n, vector<pair<int, ll> > *adj ) {
  vector<ll> dist( n + 5, INF );
  priority_queue<pair<ll, int> > q;
  q.push( { dist[start] = 0, start } );
  while( !q.empty() ) {
    int u = q.top().second;
    ll d = -q.top().first; q.pop();
    if( d > dist[u] ) continue;
    for( pair<int, ll> pv : adj[u] ) {
      int v = pv.first, w = pv.second;
      if( dist[u] + w < dist[v] )
        q.push( { -( dist[v] = dist[u] + w ), v } );
    }
  }
  return dist;
}
```

3.2 SPFA

```
// Shortest path faster algorithm avg O(E), worst case O(VE)
#define ll long long
#define INF 0x3f3f3f3f3f3f3f3f
vector<ll> spfa( int start, int n, vector<pair<int, int> > *adj ) {
  vector<ll> dist( n+5, INF );
  vector<int> pre( n+5, -1 );
  bool inQueue[MAX_N]={};
  dist[start] = 0;
  list<int> q;
  q.push_back( start );
  inQueue[start] = 1;
  while( !q.empty() ) {
    int v = q.front();
    q.pop_front();
    inQueue[v] = 0;
    for( auto p : adj[v] ) {
      int u = p.first;
      ll d = dist[v] + p.second;
      if( d < dist[u] ) {
        dist[u] = d, pre[u] = v;
        if( !inQueue[u] ) {
          if (q.size() && d < dist[q.front()]) q.push_front(u);
          else q.push_back(u);
          inQueue[u] = 1;
        }
      }
    }
  }
  return dist;
}
```

3.3 Floyd-Warshall Shortest path

```
// Shortest path O(n^3) adjacency matrix with weights and INF when no weight
#define ll long long
#define INF 0x3f3f3f3f3f3f3f3f
void fw( int n, vector<vector<ll> > &d ) {
  for( int k = 0 ; k < n ; ++k )
    for( int i = 0 ; i < n ; ++i )
      for( int j = 0 ; j < n ; ++j )
        d[i][j] = min( d[i][j], d[i][k] + d[k][j] );
}
```

3.4 Diameter

```
// start d with INF, only works with unweighted
// run bfs on all vertices O(n*m)

int d[MAXN][MAXN];
int diam;
void bfs( int s ) {
  queue<int> q;
  q.push( s );
  d[s][s] = 0;
  while( !q.empty() ) {
    int u = q.front(); q.pop();
    for( int v : g[u] ) {
      if( d[s][v] == INF ){
        d[s][v] = d[v][s] = min( d[s][u] + 1, d[v][s] );
        diam = max( d[s][u], diam );
        q.push( v );
      }
    }
  }
}

// on tree O(n+m)
#define INF 0x3f3f3f3f
int vis[MAXN];
vector<int> g[MAXN];
int t = 1;

void dfs( int u, int c, int &mc, int &x ){
  vis[u] = t;
  c++;
  for( int v : g[u] ) {
    if( vis[v] != t ) {
      if( c >= mc ) mc = c, x = v;
      dfs( v, c, mc, x );
    }
  }
}

int diameter(){
  int diam = -INF, x = -1;
  dfs( 1, 0, diam, x );
  ++t;
  dfs( x, 0, diam, x );
  return diam;
}
```

3.5 Tarjan

```cpp
// O(n+m) | index 1
int n;
vector<int> adj[MAXN];
int scc[MAXN], sccnum = 0;
int in[MAXN], low[MAXN], t = 0;
stack<int> s;
bool instack[MAXN];

void dfs( int u ) {
  low[u] = in[u] = t++;
  s.push( u );
  instack[u] = true;
  for( int v : adj[u] )
    if( in[v] == -1 )
      dfs( v ),
      low[u] = min( low[u], low[v] );
    else if( instack[v] )
      low[u] = min(low[u], in[v]);
  if( low[u] == in[u] ) {
    while( true ) {
      int su = s.top();
      s.pop();
      scc[su] = sccnum;
      instack[su] = false;
      if (su == u) break;
    }
    ++sccnum;
  }
}

void tarjan() {
  memset( scc, -1, sizeof scc );
  memset( in, -1, sizeof in );
  for( int i = 1 ; i <= n ; ++i ) if (scc[i] == -1) dfs(i);
}
```

3.6 Kosaraju

```cpp
//index 1
// O(n+m)
vector<int> adj[MAXN], adjt[MAXN];
int ord[MAXN], ordn, scc[MAXN], sccn, vis[MAXN];

void dfs( int u ) {
  vis[u] = 1;
  for( int v : adj[u] ) if ( !vis[v] ) dfs( v );
  ord[ordn++] = u;
}

void dfst( int u ) {
  vis[u] = 0;
  for( int v : adjt[u] ) if( vis[v] ) dfst( v );
  scc[u] = sccn;
}

//use:
sccn = ordn = 1;
for( int i = 1 ; i <= n ; ++i ) if( !vis[i] ) dfs( i );
for( int i = n ; i > 0 ; --i ) if( vis[ord[i]] ) dfst( ord[i] ), ++sccn;
```

3.7 LCA fast query

```cpp
// O(1) query, O(n*log n) build | index 1 | rmqb( dfs() ) to run it
#define ll long long
#define pii pair<int, int>
int tim[MAXN]; // filled with invalid time (-1)
ll dist[MAXN]; // filled with 0
vector<vector<pii> > jmp;
vector<vector<pii> > g;
int n; //vertex count

vector<pii> dfs() {
  memset( tim, -1, sizeof( tim ) );
  vector<tuple<int, int, int, ll > > q;
  q.emplace_back( 1, 0, 0, 0 );
  vector<pii> ret;
  int T = 0, v, p, d;
  ll di;
  while( !q.empty() ) {
    tie( v, p, d, di ) = q.back(); q.pop_back();
    if( d ) ret.emplace_back( d, p );
    tim[v] = T++;
    dist[v] = di;
    for( auto& e : g[v] )
      if( e.first != p )
        q.emplace_back( e.first, v, d + 1, di + e.second );
  }
  return ret;
}

void rmqb( const vector<pii>& v ) {
  int n = v.size(), depth = 31 - __builtin_clz( n ) + 1;
  jmp.assign( depth + 1, v );
  for( int i = 0 ; i < depth ; ++i )
    for( int j = 0 ; j < n ; ++j )
      jmp[i+1][j] = min( jmp[i][j], jmp[i][min( n - 1, j + ( 1 << i ) )] );
}

pii rmqq( int a, int b ) {
  int dep = 31 - __builtin_clz( b - a );
  return min( jmp[dep][a], jmp[dep][b - ( 1 << dep )] );
}

int lca( int a, int b ) {
  if( a == b ) return a;
  a = tim[a], b = tim[b];
  return rmqq( min( a, b ), max( a, b ) ).second;
}

ll distance( int a, int b ) {
  int l = lca( a, b );
  return dist[a] + dist[b] - 2 * dist[l];
}
```

3.8 LCA log query

```cpp
// To compute minimum just use the commented code | index 0
// O(log n) query | O(n log n) build
typedef pair<int,int> pii;
int parent[MAXN], level[MAXN], dist[MAXN];
int anc[MAXN][MAXLG];//, mnn[MAXM][30];
vector<pii> g[MAXN];

void dfs( int u ) {
  for( pii pv : g[u] ) {
    int v = pv.first, w = pv.second;
    if( v != parent[u] ) {
      parent[v] = u;
      level[v] = level[u] + 1;
      dist[v] = dist[u] + w;
      dfs( v );
```

```
      }
    }
  }

  void build() {
    parent[0] = level[0] = dist[0] = 0;
    dfs( 0 );
    for( int i = 0; i < n; ++i ) anc[i][0] = parent[i];//, mnn[i][0] = dist[i];
    for( int j = 1; j < MAXLG ; ++j )
      for( int i = 0; i < n; ++i ){
        anc[i][j] = anc[anc[i][j - 1]][j - 1];
        //mnn[i][j] = min( mnn[i][j - 1], mnn[anc[i][j - 1]][j - 1] );
      }
  }

  //true if v is ancestor of u
  bool is_ancestor( int u, int v ) {
    if( level[u] < level[v] ) return false;
    int d = level[u] - level[v];
    for( int i = 0 ; i < MAXLG ; ++i )
      if( d & (1<<i) ) u = anc[u][i];
    return u == v;
  }

  int lca( int u, int v ) {
    if( level[u] < level[v] ) swap( u, v );
    for( int i = MAXLG - 1; i >= 0; --i )
      if( level[u] - ( 1 << i ) >= level[v] )
        //mn = min( mn, mnn[u][i] ),
        u = anc[u][i];
    if( u == v ) return u; //return mn;
    for( int i = MAXLG - 1 ; i >= 0 ; --i )
      if( anc[u][i] != anc[v][i] )
        //mn = min( mn, min( mnn[u][i], mnn[v][i] ) ),
        u = anc[u][i], v = anc[v][i];
    return anc[u][0];
    //return min( mn, min( mnn[u][0], mnn[v][0] ) );
  }
```

3.9 Kuhn bipartite matching

```
  // Maximum cardinality (bipartite matching) O(n^3) worst case
  // if slow random_shuffle vertice orders.
  // Apply it only on left set. indexed 1
  vector<int> g[MAXN];
  int vis[MAXN], ma[MAXN], mb[MAXM];
  int n, x; // n is size of left set

  bool dfs( int u ) {
    for( int v : g[u] ) if(vis[v] != x) {
      vis[v] = x;
      if( mb[v] == -1 || dfs( mb[v] ) ) {
        mb[v] = u, ma[u] = v;
        return 1;
      }
    }
    return 0;
  }

  int kuhn() {
    memset(ma, -1, sizeof(ma));
    memset(mb, -1, sizeof(mb));
    bool aux = 1;
    int ans = 0;
    while( aux ) {
      ++x, aux = 0;
      for( int i = 1 ; i <= n ; ++i )
        if( ma[i] == -1 && dfs(i) ) ++ans, aux = 1;
    }
    return ans;
  }
```

3.10 Hopcroft-Karp Fast bipartite matching

```
  // Fast bipartite matching O(sqrt(V) * E) // indexed in 1
  int N; // size of left set
  vector<int> g[MAX_N];
  int b[MAX_N];
  int dist[MAX_N];

  bool bfs() {
    queue<int> q;
    memset( dist, -1, sizeof dist );
    for( int i = 1 ; i <= N ; ++i )
      if( b[i] == -1 )
        q.push( i ), dist[i] = 0;
    bool reached = false;
    while( !q.empty() ) {
      int n = q.front();
      q.pop();
      for( int v : g[n] ) {
        if( b[v] == -1 ) reached = true;
        else if( dist[b[v]] == -1 ) {
          dist[b[v]] = dist[n] + 1;
          q.push( b[v] );
        }
      }
    }
    return reached;
  }

  bool dfs( int n ) {
    if( n == -1 ) return true;
    for( int v : g[n] ) {
      if( b[v] == -1 || dist[b[v]] == dist[n] + 1 ) {
        if( dfs( b[v] ) ) {
          b[v] = n, b[n] = v;
          return true;
        }
      }
    }
    return false;
  }

  int hk()
  {
    memset( b, -1, sizeof b );
    int ans = 0;
    while( bfs() ) {
      for( int i = 1 ; i <= N ; ++i )
        if( b[i] == -1 && dfs( i ) ) ++ans;
    }
    return ans;
  }
```

3.11 Matrix matching

```
  // Bipartite matching O( VE ) ; w[i][j] = edge between left i and right j
  // mr, mc are match row and column
  bool match( int i, vector<vector<int> > w, int *mr, int *mc, int *vis, int x ) {
    for( int j = 0 ; j < w[i].size() ; ++j ) {
```

```
      if( w[i][j] && vis[j] != x ) {
        vis[j] = x;
        if( mc[j] < 0 || match( mc[j], w, mr, mc, vis, x ) ) {
          mr[i] = j, mc[j] = i;
          return true;
        }
      }
    }
    return false;
  }

  int bi( vector<vector<int> > w ) {
    int vis[MAX_N] = {};
    int mr[MAX_N];
    int mc[MAX_N];
    int x = 0;
    int ct = 0;

    memset( mr, -1, sizeof( mr ) );
    memset( mc, -1, sizeof( mc ) );

    for( int i = 0; i < w.size(); ++i )
      if( match( i, w, mr, mc, vis, ++x ) ) ++ct;
    return ct;
  }
```

3.12 Edmond's blossom general matching

```
  // Edmond's Blossom (general graph matching) O(VE) / pass MAX_N into constructor
  #define INV_PAIR { -1, -1 }
  struct Bloss {
    vector<vector<int> > adj;
    vector<int> pairs, fst, que;
    vector<pair<int, int> > lbl;
    int head, tail;

    Bloss( int n ) : adj( n ), pairs( n + 1, n ), fst( n + 1, n ), que( n ), lbl(
         n + 1, INV_PAIR ) {}

    void add( int u, int v ) {
      adj[u].push_back( v ), adj[v].push_back( u );
    }
    void rem( int v, int w ) {
      int t = pairs[v]; pairs[v] = w;
      if( pairs[t] != v ) return;
      if( lbl[v].second == -1 )
        pairs[t] = lbl[v].first, rem( pairs[t], t );
      else
        rem( lbl[v].first, lbl[v].second ), rem( lbl[v].second, lbl[v].first );
    }

    int find( int u ) {
      return lbl[fst[u]].first < 0 ? fst[u] : fst[u] = find( fst[u] );
    }

    void rel( int x, int y ) {
      int r = find( x );
      int s = find( y );
      if( r == s ) return;
      auto h = lbl[r] = lbl[s] = { ~x, y };
      int join;
      while( true ) {
        if( s != adj.size() ) swap( r, s );
        r = find( lbl[pairs[r]].first );
        if( lbl[r] == h ) {
          join = r; break;
        }
        else lbl[r] = h;
      }
      for( int v : { fst[x], fst[y] } ) {
        for( ; v != join ; v = fst[lbl[pairs[v]].first] ) {
          lbl[v] = { x, y };
          fst[v] = join;
          que[tail++] = v;
        }
      }
    }

    bool aug( int u ) {
      lbl[u] = { adj.size(), -1 };
      fst[u] = adj.size();
      head = tail = 0;
      for( que[tail++] = u ; head < tail ; ) {
        int x = que[head++];
        for( int y : adj[x] ) {
          if( pairs[y] == adj.size() && y != u ) {
            pairs[y] = x;
            rem( x, y );
            return true;
          }
          else if( lbl[y].first >= 0 ) rel( x, y );
          else if( lbl[pairs[y]].first == -1 ) {
            lbl[pairs[y]].first = x;
            fst[pairs[y]] = y;
            que[tail++] = pairs[y];
          }
        }
      }
      return false;
    }

    int match() {
      int ans = head = tail = 0;
      for( int u = 0 ; u < adj.size() ; ++u ) {
        if( pairs[u] < adj.size() || !aug( u ) ) continue;
        ++ans;
        for( int i = 0 ; i < tail ; ++i )
          lbl[que[i]] = lbl[pairs[que[i]]] = INV_PAIR;
        lbl[adj.size()] = INV_PAIR;
      }
      return ans;
    }
  };
```

3.13 Bridges and articulation points

```
  // return number of bridges at variable "bridges", also dp[u] calculates back
       edges from u to ancestor.
  // O(n+m) | start lvl[root] = 1
  int bridges, n, m;
  vector<pair<int, int> > g[MAXN];
  int lvl[MAXN];
  int dp[MAXN];

  void dfs( int u ){
    dp[u] = 0;
    for( pair<int, int> pv : g[u] ){
      int v = pv.first, e = pv.second;
      if( !lvl[v] ){
        lvl[v] = lvl[u] + 1;
        dfs( v );
        dp[u] += dp[v];
      }
      else if( lvl[v] < lvl[u] ) ++dp[u];
```

```
    else if( lvl[v] > lvl[u] ) --dp[u];
  }
  --dp[u];
  if( lvl[u] > 1 && !dp[u] ) ++bridges;
}

// articulation points O(n+m) index 0
int par[MAXN], art[MAXN], low[MAXN], num[MAXN], ch[MAXN], cnt;

void articulation(int u) {
  low[u] = num[u] = ++cnt;
  for (int v : adj[u]) {
    if (!num[v]) {
      par[v] = u; ++ch[u];
      articulation(v);
      if (low[v] >= num[u]) art[u] = 1;
      if (low[v] > num[u]){
        // u-v bridge
      }
      low[u] = min(low[u], low[v]);
    }
    else if (v != par[u]) low[u] = min(low[u], num[v]);
  }
}

for (int i = 0; i < n; ++i) if (!num[i])
  articulation(i), art[i] = ch[i]>1;
```

3.14 Dinic max flow

```
/* Max flow algorithm
 * Time Complexity:
 *   - O(V^2 E) for general graphs, but in practice ~O(E^1.5)
 *   - O(sqrt(V) * E) for bipartite matching
 *   - O(min(V^(2/3), E^(1/2)) E) for unit capacity graphs
 */
#define ll long long
class max_flow {
  static const ll INF = numeric_limits<ll>::max();

  struct edge {
    int t;
    unsigned long rev;
    ll cap, f;
  };

  vector<edge> adj[MAX_N];
  int dist[MAX_N];
  int ptr[MAX_N];

  bool bfs( int s, int t ) {
    memset( dist, -1, sizeof dist );
    dist[s] = 0;
    queue<int> q( { s } );
    while( !q.empty() && dist[t] == -1 ) {
      int n = q.front();
      q.pop();
      for( edge& e : adj[n] ) {
        if( dist[e.t] == -1 && e.cap != e.f ) {
          dist[e.t] = dist[n] + 1;
          q.push( e.t );
        }
      }
    }
    return dist[t] != -1;
  }

  ll aug( int n, ll amt, int t ) {
    if( n == t ) return amt;
    for( ; ptr[n] < adj[n].size() ; ++ptr[n] ) {
      edge& e = adj[n][ptr[n]];
      if( dist[e.t] == dist[n] + 1 && e.cap != e.f ) {
        ll flow = aug( e.t, min( amt, e.cap - e.f ), t );
        if( flow != 0 ) {
          e.f += flow;
          adj[e.t][e.rev].f -= flow;
          return flow;
        }
      }
    }
    return 0;
  }

public:
  void add( int u, int v, ll cap=1, ll rcap=0 ) {
    adj[u].push_back({ v, adj[v].size(), cap, 0 });
    adj[v].push_back({ u, adj[u].size() - 1, rcap, 0 });
  }

  ll calc( int s, int t ) {
    ll flow = 0;
    while( bfs( s, t ) ) {
      memset( ptr, 0, sizeof ptr );
      while( ll df = aug( s, INF, t ) ) flow += df;
    }
    return flow;
  }

  void clear() {
    for( int n = 0 ; n < MAX_N ; ++n ) adj[n].clear();
  }
};

int cut[MAXN];
void dfs( int u, max_flow &mf ) {
  cut[u] = true;
  for( auto &e : mf.adj[u] )
    if( e.cap > e.f && !cut[e.t] ) dfs( e.t, mf );
}
```

3.15 Edmonds-karp maxflow

```
// prefer index 0, O(n*m^2)
#define MAXN 55
#define INF 0x3f3f3f3f
int n, m;
int capacity[MAXN][MAXN];
vector<int> adj[MAXN];

int bfs(int s, int t, vector<int>& parent) {
    fill(parent.begin(), parent.end(), -1);
    parent[s] = -2;
    queue<pair<int, int>> q;
    q.push({s, INF});

    while (!q.empty()) {
        int cur = q.front().first;
        int flow = q.front().second;
        q.pop();

        for (int next : adj[cur]) {
            if (parent[next] == -1 && capacity[cur][next]) {
                parent[next] = cur;
                int new_flow = min(flow, capacity[cur][next]);
```

```cpp
                if (next == t)
                    return new_flow;
                q.push({next, new_flow});
            }
        }
    }

    return 0;
}

int maxflow(int s, int t) {
    int flow = 0;
    vector<int> parent(n+1);
    int new_flow;

    while (new_flow = bfs(s, t, parent)) {
        flow += new_flow;
        int cur = t;
        while (cur != s) {
            int prev = parent[cur];
            capacity[prev][cur] -= new_flow;
            capacity[cur][prev] += new_flow;
            cur = prev;
        }
    }

    return flow;
}
```

3.16 Min cost Max flow

```cpp
/* Minimum-Cost, Maximum-Flow solver using Successive Shortest Paths with
    Dijkstra and SPFA-SLF.
 * Requirements:
 *   - No duplicate or antiparallel edges with different costs.
 *   - No negative cycles.
 * Time Complexity: O(Ef lg V) average-case, O(VE + Ef lg V) worst-case.
 */
#define INF 0x3f3f3f3f3f3f3f3f
template<int V, class T=long long>
class mcmf {
  unordered_map<int, T> cap[V], cost[V];
  T dist[V];
  int pre[V];
  bool visited[V];
  void spfa(int s) {
    static list<int> q;
    memset(pre, -1, sizeof pre);
    fill(dist, dist+V, INF);
    memset(visited, 0, sizeof visited);
    dist[s] = 0;
    q.push_back(s);
    while (!q.empty()) {
      int v = q.front();
      q.pop_front();
      visited[v] = false;
      for (auto p : cap[v]) if (p.second) {
        int u = p.first;
        T d = dist[v] + cost[v][u];
        if (d < dist[u]) {
          dist[u] = d, pre[u] = v;
          if (!visited[u]) {
            if (q.size() && d < dist[q.front()]) q.push_front(u);
            else q.push_back(u);
            visited[u] = true;
          }
        }
      }
    }
  }
  void dijkstra(int s) {
    static priority_queue<pair<T, int>, vector<pair<T, int> >,
        greater<pair<T, int> > > pq;
    memset(pre, -1, sizeof pre);
    fill(dist, dist+V, INF);
    memset(visited, 0, sizeof visited);
    dist[s] = 0;
    pq.push({0, s});
    while (!pq.empty()) {
      int v = pq.top().second;
      pq.pop();
      if (visited[v]) continue;
      visited[v] = true;
      for (auto p : cap[v]) if (p.second) {
        int u = p.first;
        T d = dist[v] + cost[v][u];
        if (d < dist[u]) {
          dist[u] = d, pre[u] = v;
          pq.push({d, u});
        }
      }
    }
  }
  void reweight() {
    for (int v = 0; v < V; v++) {
      for (auto& p : cost[v]) {
        p.second += dist[v] - dist[p.first];
      }
    }
  }
public:
  unordered_map<int, T> flows[V];
  void add(int u, int v, T f=1, T c=0) {
    cap[u][v] += f;
    cost[u][v] = c;
    cost[v][u] = -c;
  }
  pair<T, T> calc(int s, int t) {
    spfa(s);
    T totalflow = 0, totalcost = 0;
    T fcost = dist[t];
    while (true) {
      reweight();
      dijkstra(s);
      if (~pre[t]) {
        fcost += dist[t];
        T flow = cap[pre[t]][t];
        for (int v = t; ~pre[v]; v = pre[v])
          flow = min(flow, cap[pre[v]][v]);
        for (int v = t; ~pre[v]; v = pre[v]) {
          cap[pre[v]][v] -= flow;
          cap[v][pre[v]] += flow;
          flows[pre[v]][v] += flow;
          flows[v][pre[v]] -= flow;
        }
        totalflow += flow;
        totalcost += flow * fcost;
      }
      else break;
    }
    return { totalflow, totalcost };
  }
  void clear() {
    for (int i = 0; i < V; i++) {
      cap[i].clear();
      cost[i].clear();
```

```
      flows[i].clear();
      dist[i] = pre[i] = visited[i] = 0;
    }
  }
};
```

3.17 Min cost Max flow 2

```
// index 0
#define ll long long
const ll inf = 0x3f3f3f3f3f3f3f3f;
struct edge {
  ll a, b, cap, cost, flow;
  size_t back;
};
vector<edge> e;
vector<ll> g[MAXN];
void addedge(ll a, ll b, ll cap, ll cost) {
  edge e1 = {a,b,cap,cost,0,g[b].size()};
  edge e2 = {b,a,0,-cost,0,g[a].size()};
  g[a].push_back((ll) e.size());
  e.push_back(e1);
  g[b].push_back((ll) e.size());
  e.push_back(e2);
}
ll n, s, t, m;
ll k = inf; // The maximum amount of flow allowed
// Returns {flow,cost}
pair<ll,ll> getflow() {
  ll flow = 0, cost = 0;
  while(flow < k) {
    vector<ll> id(n, 0);
    vector<ll> d(n, inf);
    vector<ll> q(n);
    vector<ll> p(n);
    vector<size_t> p_edge(n);
    ll qh=0, qt=0;
    q[qt++] = s;
    d[s] = 0;
    while(qh != qt) {
      ll v = q[qh++];
      id[v] = 2;
      if(qh == n) qh = 0;
      for(size_t i=0; i<g[v].size(); ++i) {
        edge& r = e[g[v][i]];
        if(r.flow < r.cap && d[v] + r.cost < d[r.b]) {
          d[r.b] = d[v] + r.cost;
          if(id[r.b] == 0) {
            q[qt++] = r.b;
            if(qt == n) qt = 0;
          }
          else if(id[r.b] == 2) {
            if(--qh == -1) qh = n-1;
            q[qh] = r.b;
          }
          id[r.b] = 1;
          p[r.b] = v;
          p_edge[r.b] = i;
        }
      }
    }
    if(d[t] == inf) break;
    ll addflow = k - flow;
    for(ll v=t; v!=s; v=p[v]) {
      ll pv = p[v]; size_t pr = p_edge[v];
      addflow = min(addflow, e[g[pv][pr]].cap - e[g[pv][pr]].flow);
    }
    for(ll v=t; v!=s; v=p[v]) {
      ll pv = p[v]; size_t pr = p_edge[v], r = e[g[pv][pr]].back;
      e[g[pv][pr]].flow += addflow;
      e[g[v][r]].flow -= addflow;
      cost += e[g[pv][pr]].cost * addflow;
    }
    flow += addflow;
  }
  return {flow,cost};
}
```

3.18 Maximum matching (hungarian)

```
// O(VE)
typedef long long ll;
const ll inf = 0x3f3f3f3f3f3f3f3f;

ll u[MAXN], v[MAXN];
int p[MAXN], way[MAXN];
ll minv[MAXN];
bool used[MAXN];

pair<vector<int>, ll> solve(const vector<vector<ll>> &matrix) {
  int n = matrix.size();
  if (n == 0) return {vector<int>(), 0};
  for(int i = 1; i <= n; i++) {
    for(int i = 0; i <= n; i++) minv[i] = inf;
    memset(way, 0, (n+1) * sizeof(int));
    for(int j = 0; j <= n; j++) used[j] = false;
    p[0] = i;
    int k0 = 0;
    do {
      used[k0] = true;
      int i0 = p[k0], k1;
      ll delta = inf;
      for(int j = 1; j <= n; j++) {
        if(!used[j]) {
          ll cur = matrix[i0-1][j-1] - u[i0] - v[j];
          if(cur < minv[j]) {
            minv[j] = cur;
            way[j] = k0;
          }
          if(minv[j] < delta) {
            delta = minv[j];
            k1 = j;
          }
        }
      }
      for(int j = 0; j <= n; j++) {
        if(used[j]) {
          u[p[j]] += delta;
          v[j] -= delta;
        } else {
          minv[j] -= delta;
        }
      }
      k0 = k1;
    } while (p[k0] != 0);
    do {
      int k1 = way[k0];
      p[k0] = p[k1];
      k0 = k1;
    } while (k0 != 0);
  }
  // Get actual matching
  vector<int> ans(n, -1);
```

```
  for(int j = 1; j <= n; j++) {
    if(p[j] == 0) continue;
    ans[p[j] - 1] = j-1;
  }
  return {ans, -v[0]};
}
```

3.19 Kruskal MST

```
// O(m log(m))
#define ll long long
struct edge {
  int u, v; ll w;
  edge( int _u, int _v, ll _w ) : u(_u),v(_v),w(_w) {}
  bool operator < ( const edge &o ) const {
    return w < o.w;
  }
};

vector<edge> edges;
int root[MAXN];
int n, m;

int find( int x ) { return ( x == root[x] ) ? x : root[x] = find( root[x] ); }

bool merge( int u, int v ){
  if( ( u = find( u ) ) == ( v = find( v ) ) ) return false;
  root[u] = v;
  return true;
}

ll kruskal()
{
  ll cost = 0;
  sort( edges.begin(), edges.end() );
  for( int i = 0 ; i <= n ; ++i ) root[i] = i;
  for( int i = 0 ; i < m ; ++i )
    if( merge( edges[i].u, edges[i].v ) ) cost += edges[i].w;
  return cost;
}
```

3.20 Tarjan Biconnected Components

```
// Complexity O(n+m)
int N;
vector<int> adj[MAXN];
vector<int> bcc[MAXN];
int bccnum = 0;
int in[MAXN], low[MAXN], t = 0;
stack<pair<int, int> > s;
bool visited[MAXN];

void dfs( int u, int p = -1 ) {
  visited[u] = true;
  low[u] = in[u] = t++;
  for( int v : adj[u] ) if ( v != p ) {
    if( !visited[v] ) {
      s.emplace( v, u );
      dfs( v, u );
      low[u] = min( low[u], low[v] );
      if( low[v] >= in[u] ) { // u is articulation
        while( true ) {
          auto p = s.top();
          s.pop();
          int a = p.first, b = p.second;
          if( bcc[a].empty() || bcc[a].back() != bccnum )
            bcc[a].push_back( bccnum );
          if( bcc[b].empty() || bcc[b].back() != bccnum )
            bcc[b].push_back( bccnum );
          if( a == v && b == u ) break;
        }
        ++bccnum;
      }
    }
    else if( in[v] < in[u] ) {
      low[u] = min( low[u], in[v] );
      s.emplace( v, u );
    }
  }
}

void tarjan() {
  for( int i = 1 ; i <= N ; ++i ) if ( !visited[i] ) dfs( i );
}

bool biconnected( int u, int v ) {
  for( int c : bcc[u] )
    if( binary_search( bcc[v].begin(), bcc[v].end(), c ) )
      return true;
  return false;
}
```

3.21 Centroid decomposition

```
// cpar[i] stores parent of i | O(n) | index 0
int N;
vector<int> adj[MAXN];
int sz[MAXN];
int cpar[MAXN];
bool vis[MAXN];

void dfs( int n, int p = -1 ) {
  sz[n] = 1;
  for( int v : adj[n] ) if( v != p && !vis[v] ) dfs( v, n ), sz[n] += sz[v];
}

int centroid( int n ) {
  dfs( n );
  int num = sz[n];
  int p = -1;
  do {
    int nxt = -1;
    for( int v : adj[n] ) if( v != p && !vis[v] )
      if( 2 * sz[v] > num ) nxt = v;
    p = n, n = nxt;
  } while( ~n );
  return p;
}

void decomp( int n = 0, int p = -1 ) {
  int c = centroid( n );
  vis[c] = true;
  cpar[c] = p;
  for( int v : adj[c] ) if ( !vis[v] ) decomp( v, c );
}
```

3.22 Euler tour

```
// This gives a path that each edge is visited only one time | adj[i].second is
    the edge id
// It has an euler cycle iff all vertex have even degree | O(n+m)
int N, M;
vector<pair<int, int> > adj[MAXN];
int cur[MAXN];
bool used[MAXM];
vector<int> tour;

void dfs( int n ) {
  while( cur[n] != adj[n].size() ) {
    if( used[adj[n][cur[n]].second] ) {
      ++cur[n];
      continue;
    }
    auto p = adj[n][cur[n]++];
    used[p.second] = true;
    dfs( p.first );
  }
  tour.push_back( n );
}
```

3.23 Hierholzers(euler circuit)

```
// Euler circuit for directed graphs O(n+m)
// example output 0 -> 1 -> 2 ... -> 0
// index 0
vector<int> circuit( vector<vector<int> > adj ){
  unordered_map<int,int> edge_count;
  for( int i = 0 ; i < adj.size() ; ++i ){
    edge_count[i] = adj[i].size();
  }
  if( !adj.size() ) return;
  stack<int> curr_path;
  vector<int> circuit;
  curr_path.push( 0 );
  int curr_v = 0;
  while( !curr_path.empty() ){
    if( edge_count[curr_v] ){
      curr_path.push(curr_v);
      int next_v = adj[curr_v].back();
      edge_count[curr_v]--;
      adj[curr_v].pop_back();
      curr_v = next_v;
    } else {
      circuit.push_back(curr_v);
      curr_v = curr_path.top();
      curr_path.pop();
    }
  }
  return circuit;
}
```

3.24 Min cut Stoer-Wagner

```
// a is adjacency matrix bidirected
// minimum cut problem in undirected weighted graphs with non-negative weights
// O(VE)
memset(use,0,sizeof(use));
ans=maxlongint;
for (int i=1;i<N;i++)
{
  memcpy(visit,use,505*sizeof(int));
  memset(reach,0,sizeof(reach));
  memset(last,0,sizeof(last));
  t=0;
  for (int j=1;j<=N;j++)
    if (use[j]==0) {t=j;break;}
  for (int j=1;j<=N;j++)
    if (use[j]==0) reach[j]=a[t][j],last[j]=t;
  visit[t]=1;
  for (int j=1;j<=N-i;j++)
  {
    maxc=maxk=0;
    for (int k=1;k<=N;k++)
      if ((visit[k]==0)&&(reach[k]>maxc)) maxc=reach[k],maxk=k;
    c2=maxk,visit[maxk]=1;
    for (int k=1;k<=N;k++)
      if (visit[k]==0) reach[k]+=a[maxk][k],last[k]=maxk;
  }
  c1=last[c2];
  sum=0;
  for (int j=1;j<=N;j++)
    if (use[j]==0) sum+=a[j][c2];
  ans=min(ans,sum);
  use[c2]=1;
  for (int j=1;j<=N;j++)
    if ((c1!=j)&&(use[j]==0)) {a[j][c1]+=a[j][c2];a[c1][j]=a[j][c1];}
}
```

3.25 AHU Isomorphic tree

```
// Yes if both trees are isomorphic | Index 1 | O(nlogn)
typedef vector<int> vi;
int n, a, b;
vi adj[2][MAXN];
int vis[MAXN], p[MAXN], sz[MAXN], x;
vi centr[2];
map<map<int, int>, int> m;
void dfsc(int t, int u) {
  vis[u] = x;
  sz[u] = 1;
  int ok = 1;
  for (int v : adj[t][u]) {
    if (v == p[u]) continue;
    if (vis[v] != x) p[v]=u, dfsc(t, v);
    sz[u] += sz[v];
    if (sz[v] > n/2) ok=0;
  }
  if (n-sz[u] > n/2) ok=0;
  if (ok) centr[t].push_back(u);
}
int dfs(int t, int u) {
  vis[u]=x;
  map<int, int> c;
  for (int v : adj[t][u]) {
    if (v == p[u]) continue;
    if (vis[v] != x) p[v]=u, dfs(t, v);
    c[sz[v]]++;
  }
  if (!m.count(c)) m[c] = m.size();
  return sz[u]=m[c];
}

// This goes on Main
int es[2];
for( int j = 0 ; j < 2 ; ++j ) {
  ++x;
  p[1] = -1;
  dfsc(j, 1);
  ++x;
```

```
    p[centr[j][0]] = -1;
    es[j] = dfs(j, centr[j][0]);
  }
  es[0] = es[0] == es[1];
  if (!es[0] && centr[0].size()>1) {
    ++x, p[centr[0][1]]=-1;
    es[0] = dfs(0, centr[0][1]) == es[1];
  }
  puts( ( es[0] ? "YES" : "NO" ) );
```

3.26 Prufer code

```
// the number of labeled trees is n^{n-2}.
// O(n)

int n;
vector<int> adj[MAXN];

void addEdge(int u, int v) {
  adj[u].push_back(v);
  adj[v].push_back(u);
}


vector<int> treeToCode() {
  vector<int> deg(n), parent(n, -1), code;
  function<void(int)> dfs = [&](int u) {
    deg[u] = adj[u].size();
    for (int v: adj[u]) {
      if (v != parent[u]) {
        parent[v] = u;
        dfs(v);
      }
    }
  };
  dfs(n-1);

  int index = -1;
  while (deg[++index] != 1);
  for (int u = index, i = 0; i < n-2; ++i) {
    int v = parent[u];
    code.push_back(v);
    if (--deg[v] == 1 && v < index) {
      u = v;
    } else {
      while (deg[++index] != 1);
      u = index;
    }
  }
  return code;
}

Tree codeToTree(vector<int> code) {
  int n = code.size() + 2;
  Tree T(n);
  vector<int> deg(n, 1);
  for (int i = 0; i < n-2; ++i)
    ++deg[code[i]];

  int index = -1;
  while (deg[++index] != 1);
  for (int u = index, i = 0; i < n-2; ++i) {
    int v = code[i];
    addEdge(u, v);
    --deg[u]; --deg[v];
    if (deg[v] == 1 && v < index) {
      u = v;
    } else {
      while (deg[++index] != 1);
      u = index;
    }
  }
  for (int u = 0; u < n-1; ++u)
    if (deg[u] == 1)
      addEdge(u, n-1);
  return T;
}
```

3.27 2-Sat

```
// 2-SAT - O(V+E)
// For each variable x, we create two nodes in the graph: u and !u
// If the variable has index i, the index of u and !u are: 2*i and 2*i+1
// Adds a statment u => v
void add(int u, int v){
  adj[u].pb(v);
  adj[v^1].pb(u^1);
}

//0-indexed variables; starts from var_0 and goes to var_n-1
for( int i = 0; i < n; ++i ){
  tarjan(2*i), tarjan(2*i + 1);
  //scc is a tarjan variable that says the component from a certain node
  if( scc[2*i] == scc[2*i + 1] ) //Invalid
  if( scc[2*i] < scc[2*i + 1] ) //Var_i is true
  else //Var_i is false

  //its just a possible solution!
}
```

3.28 Traveling salesman problem

```
// Find hamiltonian cycle with minimum weight
// change to commented in order to solve hamiltonian path
// O(2^n * n^2 )
// index 0
int n;
int dist[MAXN][MAXN];

int TSP(){
  int dp[1 << n][n];
  memset( dp, INF, sizeof( dp ) );
  dp[1][0] = 0;// for( int i = 0 ; i < n ; ++i ) dp[1<<i][i] = 0;
  for( int mask = 1 ; mask < 1 << n ; mask += 2 ) // mask = 0, ++mask
    for( int i = 1 ; i < n ; ++i )// i from 0
    if( ( mask & 1 << i ) != 0 )
      for( int j = 0 ; j < n ; ++j )
        if( ( mask & 1 << j ) != 0 )
          dp[mask][i] = min( dp[mask][i], dp[mask ^ (1 << i)][j] + dist[j][i] );
  int res = INF;
  for( int i = 1 ; i < n ; ++i )
    // min( res, dp[(1<<n)-1][i] )
    res = min( res, dp[(1 << n) - 1][i] + dist[i][0] );
  // reconstruct path
  int cur = (1 << n) - 1;
  int order[n];
  int last = 0;
  for( int i = n - 1 ; i >= 1 ; --i ) {// i>=0
    int bj = -1;
    for( int j = 1 ; j < n ; ++j ) {//j=0
      if( ( cur & 1 << j ) != 0 &&
//( bj==-1 ||
```

```
      //dp[cur][bj] + (last == -1 ? 0 : dist[bj][last] > dp[cur][j] + (last == -1 ? 0
           : dist[j][last] ) )
          ( bj == -1 || dp[cur][bj] + dist[bj][last] > dp[cur][j] + dist[j][last]
              ) ) bj = j;
        order[i] = bj;
        cur ^= 1 << bj;
        last = bj;
      }
      return res;
    }
  }

  // O(n^2) with Ore condition d(u) + d(v) >= n, (u,v) not in E.
  vector<int> hamilton_cycle() {
    auto X = [&](int i) { return i < n ? i : i - n; }; // faster than mod
    vector<int> cycle(n);
    iota(cycle.begin(), cycle.end(), 0);
    while (1) {
      bool updated = false;
      for (int i = 0; i < n; ++i) {
        if (adj[cycle[i]].count(cycle[X(i+1)])) continue;
        for (int j = i+2; j < i+n; ++j) {
          if (adj[cycle[i]].count(cycle[X(j)]) &&
            adj[cycle[X(i+1)]].count(cycle[X(j+1)])) {
            for (int k = i+1, l = j; k < l; ++k, --l)
              swap(cycle[X(k)], cycle[X(l)]);
            updated = true;
            break;
          }
        }
      }
      if (!updated) break;
    }
    return cycle;
  }
```

3.29 Chromatic Number

```
// index 0
// O(2^n * n)
int n;
vector<int> adj[MAXN];

int chromaticNumber() {
  const int N = 1 << n;
  vector<int> nbh(n);
  for (int u = 0; u < n; ++u)
    for (int v: adj[u])
      nbh[u] |= (1 << v);

  int ans = n;
  for( int d: {7} ) { // ,11,21,33,87,93}) {
    long long mod = 1e9 + d;
    vector<long long> ind(N), aux(N, 1);
    ind[0] = 1;
    for (int S = 1; S < N; ++S) {
      int u = __builtin_ctz(S);
      ind[S] = ind[S^(1<<u)] + ind[(S^(1<<u))&~nbh[u]];
    }
    for (int k = 1; k < ans; ++k) {
      long long chi = 0;
      for (int i = 0; i < N; ++i) {
        int S = i ^ (i >> 1); // gray-code
        aux[S] = (aux[S] * ind[S]) % mod;
        chi += (i & 1) ? aux[S] : -aux[S];
      }
      if (chi % mod) ans = k;
```

```
      }
    }
    return ans;
  }
```

3.30 Dynamic reachability in DAG

```
// It is a data structure that admits the following operations:
//  add_edge(s, t): insert edge (s,t) to the network if
//                  it does not make a cycle
// is_reachable(s, t): return true iff there is a path s --> t
// amortized O(n) per update

struct dag_reachability {
  int n;
  vector<vector<int>> parent;
  vector<vector<vector<int>>> child;
  dag_reachability(int n) : n(n), parent(n, vector<int>(n, -1)),
    child(n, vector<vector<int>>(n)) { }
  bool is_reachable(int src, int dst) {
    return src == dst || parent[src][dst] >= 0;
  }
  bool add_edge(int src, int dst) {
    if (is_reachable(dst, src)) return false; // break DAG condition
    if (is_reachable(src, dst)) return true; // no-modification performed
    for (int p = 0; p < n; ++p)
      if (is_reachable(p, src) && !is_reachable(p, dst))
        meld(p, dst, src, dst);
    return true;
  }
  void meld(int root, int sub, int u, int v) {
    parent[root][v] = u;
    child[root][u].push_back(v);
    for (int c: child[sub][v])
      if (!is_reachable(root, c))
        meld(root, sub, v, c);
  }
};
```

3.31 K-ShortestPaths

```
// We are given a weighted graph. The k-shortest walks problem
// seeks k different s-t walks (paths allowing repeated vertices)
// in the increasing order of the lengths.
// O(m log m) construction
// O(k log k) for k-th search
struct Graph {
  int n, m = 0;
  vector<int> head;
  vector<int> src, dst, next, prev;

  using Weight = long long;
  vector<Weight> weight;
  Graph(int n) : n(n), head(n, -1) { }
  int addEdge(int u, int v, Weight w) {
    next.push_back(head[u]);
    src.push_back(u);
    dst.push_back(v);
    weight.push_back(w);
    return head[u] = m++;
  }
};
constexpr Graph::Weight INF = 1e15;
struct KShortestWalks {
```

```
Graph g;
vector<Graph::Weight> dist;
vector<int> tree, order;
void reverseDijkstra(int t) {
  vector<vector<int>> adj(g.n);
  for (int u = 0; u < g.n; ++u)
    for (int e = g.head[u]; e >= 0; e = g.next[e])
      adj[g.dst[e]].push_back(e);
  dist.assign(g.n, INF);
  tree.assign(g.n, ~g.m);
  using Node = tuple<Graph::Weight,int>;
  priority_queue<Node, vector<Node>, greater<Node>> que;
  que.push(make_tuple(0, t));
  dist[t] = 0;
  while (!que.empty()) {
    int u = get<1>(que.top()); que.pop();
    if (tree[u] >= 0) continue;
    tree[u] = ~tree[u];
    order.push_back(u);
    for (int e: adj[u]) {
      int v = g.src[e];
      if (dist[v] > dist[u] + g.weight[e]) {
        tree[v] = ~e;
        dist[v] = dist[u] + g.weight[e];
        que.push(Node(dist[v], v));
      }
    }
  }
}
struct Node { // Persistent Heap (Leftist Heap)
  int e;
  Graph::Weight delta;
  Node *left = 0, *right = 0;
  int rnk = 0;
} *root = 0;
static Node *merge(Node *x, Node *y) {
  if (!x) return y;
  if (!y) return x;
  if (x->delta > y->delta) swap(x, y);
  x = new Node(*x);
  x->right = merge(x->right, y);
  if (!x->left || x->left->rnk < x->rnk) swap(x->left, x->right);
  x->rnk = (x->right ? x->right->rnk : 0) + 1;
  return x;
}
vector<Node*> deviation;
void buildHeap() {
  deviation.resize(g.n);
  for (int u: order) {
    int v = -1;
    for (int e = g.head[u]; e >= 0; e = g.next[e]) {
      if (tree[u] == e) v = g.dst[e];
      else if (dist[g.dst[e]] < INF) {
        auto delta = g.weight[e] - dist[g.src[e]] + dist[g.dst[e]];
        deviation[u] = merge(deviation[u], new Node({e, delta}));
      }
    }
    if (v >= 0) deviation[u] = merge(deviation[u], deviation[v]);
  }
}
KShortestWalks(Graph g_, int t) : g(g_) {
  reverseDijkstra(t);
  buildHeap();
}
void enumerate(int s, int kth) {
  int k = 0;
  Node *x = deviation[s];
  Graph::Weight len = dist[s];
  ++k;
  using SearchNode = tuple<Node*, Graph::Weight>;
  auto comp = [](SearchNode x, SearchNode y) { return get<1>(x) > get<1>(y);
     };
  priority_queue<SearchNode, vector<SearchNode>, decltype(comp)> que(comp);
  if (x) que.push(SearchNode(x, len + x->delta));
  while (!que.empty() && k < kth) {
    tie(x, len) = que.top(); que.pop();
    int e = x->e, u = g.src[e], v = g.dst[e];
    cout << len << endl; ++k;
    if (deviation[v]) que.push(SearchNode(deviation[v], len+deviation[v]->
        delta));
    for (Node *y: {x->left, x->right})
      if (y) que.push(SearchNode(y, len + y->delta-x->delta));
  }
  while (k < kth) { cout << -1 << endl; ++k; }
}
};
```

3.32 Functional graphs

```
// index 1
// dg[i] = degree of vertex i
// proc[i] = processed vertex on time i
// par[i] = parent of i
// sub[i] = size of subtree of vertex i
// parCycle[i] = closest vertex to i inside cycle
// depth[i] = depth of i or # of edges until parCycle[i]
// cycle[i] = index of cycle closest to i
// ini[i] = first vertex of cycle i
// sz[i] = size of cycle i
// idOnCycle[i] = id of vertex i on cycle
vector<int> proc, g[MAXN];
vector<int> cycles[MAXN];
bool vis[MAXN], onCycle[MAXN];
int par[MAXN], depth[MAXN], sub[MAXN], cycle[MAXN];
int ini[MAXN], sz[MAXN], idOnCycle[MAXN], cycleCount;
int parCycle[MAXN], n, dg[MAXN];

int findParent(int u) {
  for( int v : g[u] ) if( !vis[v] ) return v;
  return -1;
}

void foundCycle(int u){
  int iniv = u;
  int idCycle = ++cycleCount;
  int curId = 0;
  ini[idCycle] = u;
  sz[idCycle] = 0;
  cycles[idCycle].clear();
  while( vis[u] == 0 ) {
    vis[u] = 1;
    par[u] = findParent(u);
    if(par[u] == -1) par[u] = iniv;
    parCycle[u] = u, cycle[u] = idCycle;
    onCycle[u] = 1, idOnCycle[u] = curId;
    cycles[idCycle].push_back(u);
    ++sz[idCycle], ++sub[u], depth[u] = 0;
    u = par[u], ++curId;
  }
}

void lenha(){
  queue<int> q;
  for( int i = 1 ;  i<= n ; ++i )
    if(dg[i] == 1 ) q.push(i), vis[i] = 1;
  while(!q.empty()){
```

```
    int u = q.front(); q.pop();
    proc.push_back(u);
    int v = findParent(u);
    par[u] = v, ++sub[u];
    sub[v] += sub[u], --dg[v];
    if(dg[v] == 1 ) q.push(v), vis[v] = 1;
  }
  cycleCount = 0;
  for( int i = 1 ;  i<= n ; ++i )
    if(!vis[i]) foundCycle(i);
  for( int i = proc.size() - 1 ; i >= 0 ; --i ) {
    int v = proc[i], pv = par[v];
    parCycle[v] = parCycle[pv];
    cycle[v] = cycle[pv];
    onCycle[v] = 0, idOnCycle[v] = -1;
    depth[v] = depth[pv] + 1;
  }
}
```

4 Data structures

4.1 Sparse Table

```
//query from [first,last) / O( n * log(n) ) to build and O(1) to query | index 0
vector<vector<int> > jmp;
void build( const vector<int>& v ) {
  int n = v.size(), depth = 31 - __builtin_clz( N ) + 1;
  jmp.assign( depth + 1, v );
  for( int i = 0 ; i < depth ; ++i )
    for( int j = 0 ; j < n ; ++j )
      jmp[i+1][j] = min( jmp[i][j], jmp[i][min( n - 1, j + ( 1 << i ) )] );
}
int query( int a, int b ) {
  int dep = 31 - __builtin_clz( b - a );
  return min( jmp[dep][a], jmp[dep][b - ( 1 << dep )] );
}
```

4.2 Binary Indexed Tree

```
// Query range: query( r ) - query( l -1 ) | index 1 | O(log n)
#define ll long long
struct BIT {
  ll b[MAXN]={};
  ll sum( int x ) {
    ll r = 0;
    for(x += 2 ; x ; x -= x & -x ) r += b[x];
    return r;
  }
  void upd( int x, ll v ) {
    for(x += 2 ; x < MAXN ; x += x & -x ) b[x] += v;
  }
};
struct BITRange {
  BIT a,b;
  ll sum( int x ) {
    return a.sum( x ) * x + b.sum( x );
  }
  ll query( int l, int r ) {
    return sum( r ) - sum( l - 1 );
  }
  void update( int l, int r, ll v ) {
    a.upd( l, v ), a.upd( r + 1, -v );
    b.upd( l, -v*( l - 1 ) ), b.upd( r + 1, v * r );
  }
};
```

4.3 2D query sum with Treap & BIT

```
// index 1 | build: O(n^2 * log^2(n)) | query & updt: O(log^2(n))
// 3d sum query: do ( 2d with kmax ) - ( 2d with kmin )
int bit[MAXN][MAXN];

void update(int i, int j, int v) {
  for (; i < N; i+=i&-i)
    for (int jj = j; jj < N; jj+=jj&-jj)
      bit[i][jj] += v;
}

int query(int i, int j) {
  int res = 0;
  for (; i; i-=i&-i)
    for (int jj = j; jj; jj-=jj&-jj)
      res += bit[i][jj];
  return res;
}

int query(int imin, int jmin, int imax, int jmax) {
  return query(imax, jmax) - query(imax, jmin-1) - query(imin-1, jmax) + query(
      imin-1, jmin-1);
}
```

4.4 Disjoint set with persistency

```
// main: link[i] = i, rank[i] = 0, his[i] = 0; O( log n )
int link[MAXN], rank[MAXN], his[MAXN];

int find( int x, int t ) { return ( link[x] == x || his[x] > t ) ? x : find(
    link[x], t ); }

bool join( int a, int b, int t ) {
  if( ( a = find( a ) ) == ( b = find( b ) ) ) return false;
  if( rank[a] < rank[b] ) swap( a, b );
  else if( rank[a] == rank[b] ) ++rank[a];
  // bysize
  // if( size[a] < size[b] ) swap( a, b );
  // size[a] += size[b];
  link[b] = a, his[a] = t;
  return true;
}
```

4.5 MinQueue

```
// Add(x) adds x to every element in the queue
// to maxqueue change >= to <=
// O(1)
struct MinQueue {
  int plus = 0;
  int sz = 0;
  deque<pair<int, int> > dq;
  void push( int x ) {
    x -= plus;
    int amt = 1;
    while( dq.size() and dq.back().first >= x )
      amt += dq.back().second, dq.pop_back();
    dq.push_back( { x, amt } ), ++sz;
  }
```

```
  void pop() {
    --dq.front().second, --sz;
    if( !dq.front().second ) dq.pop_front();
  }
  bool empty() { return dq.empty(); }
  void clear() { plus = 0; sz = 0; dq.clear(); }
  void add( int x ) { plus += x; }
  int min() { return dq.front().first + plus; }
  int size() { return sz; }
};
```

4.6 Ordered Set

```
// find_by_order returns an iterator to the element at a given position
// order_of_key returns the position of a given element
// If the element isn't in the set, we get the position that the element would
    have
// O(log n)
#include <ext/pb_ds/assoc_container.hpp>
#include <ext/pb_ds/detail/standard_policies.hpp>
using namespace __gnu_pbds;

#include <ext/pb_ds/tree_policy.hpp>
typedef tree<int,null_type,less<int>,rb_tree_tag,
tree_order_statistics_node_update> ordered_set;

// Patricia tree implementation
#include <ext/pb_ds/trie_policy.hpp>
typedef trie< string, null_type, trie_string_access_traits<>,
pat_trie_tag, trie_prefix_search_node_update> pref_trie;
//example( ?prefix list all words with it +word add word ) 10000 limit on
    operations
while( cin >> x ){
  if( x[0] == '?' ) {
    cout << x.substr(1) << endl;
    auto range=base.prefix_range( x.substr( 1 ) );
    int t=0;
    for( auto it = range.first ; t < 20 && it != range.second ; ++it, ++t )
      cout<<"  "<<*it<<endl;
  }
  else base.insert(x.substr(1));
}
```

4.7 Lazy segment tree

```
// Index 0
// O(n log n) build | O(log n) query
// check if 0 should be returned on query (INF on max/min)
#define ll long long
ll st[MAXSEG];
ll lazy[MAXSEG];

void push(int node, int lo, int hi) {
  if (lazy[node] == 0) return;
  st[node] += lazy[node]; //(hi-lo+1)*lazy[node] for sum
  if (lo != hi) {
    lazy[2 * node + 1] += lazy[node];
    lazy[2 * node + 2] += lazy[node];
  }
  lazy[node] = 0;
}

void update(int s, int e, ll x, int lo=0, int hi=-1, int node=0) {
  if (hi == -1) hi = N - 1;
  push(node, lo, hi);
  if (hi < s || lo > e) return;
  if (lo >= s && hi <= e) {
    lazy[node] = x;
    push(node, lo, hi);
    return;
  }
  int mid = (lo + hi) / 2;
  update(s, e, x, lo, mid, 2 * node + 1);
  update(s, e, x, mid + 1, hi, 2 * node + 2);
  st[node] = max(st[2 * node + 1], st[2 * node + 2]);
}

ll query(int s, int e, int lo=0, int hi=-1, int node=0) {
  if (hi == -1) hi = N - 1;
  push(node, lo, hi);
  if (hi < s || lo > e) return -0x3f3f3f3f;
  if (lo >= s && hi <= e) return st[node];
  int mid = (lo + hi) / 2;
  return max(query(s, e, lo, mid, 2 * node + 1),
      query(s, e, mid + 1, hi, 2 * node + 2));
}
```

4.8 Persistent segment tree

```
// same as segtree, but with persistency :D
#define MAXN 100013
#define MAXLGN 18
#define MAXSEG (2 * MAXN * MAXLGN)
int N;
struct node {
  node *l, *r;
  int x;
} vals[MAXSEG]; int t = 0;
node* tree[MAXN];

node* build_tree(int lo=0, int hi=-1) {
  if (hi == -1) hi = N - 1;
  node* cur = &vals[t++];
  if (lo != hi) {
    int mid = (lo + hi) / 2;
    cur->l = build_tree(lo, mid);
    cur->r = build_tree(mid + 1, hi);
  }
  return cur;
}

node* update(node* n, int i, int x, int lo=0, int hi=-1) {
  if (hi == -1) hi = N - 1;
  if (hi < i || lo > i) return n;
  node* v = &vals[t++];
  if (lo == hi) { v->x = n->x + x; return v; }
  int mid = (lo + hi) / 2;
  v->l = update(n->l, i, x, lo, mid);
  v->r = update(n->r, i, x, mid + 1, hi);
  v->x = v->l->x + v->r->x;
  return v;
}

int query(node* n, int s, int e, int lo=0, int hi=-1) {
  if (hi == -1) hi = N - 1;
  if (hi < s || lo > e) return 0;
  if (lo >= s && hi <= e) return n->x;
  int mid = (lo + hi) / 2;
  return query(n->l, s, e, lo, mid) +
      query(n->r, s, e, mid + 1, hi);
}
```

4.9 Mergesort tree

```
// Mergesort Tree - Time <O(nlognlogn), O(nlogn)> - Memory O(nlogn)
// Mergesort Tree is a segment tree that stores the sorted subarray
// on each node. index 1

vector<int> st[4*MAXN];

void build(int p, int l, int r) {
  if( l == r ) { st[p].push_back( s[l] ); return; }
  build(2*p, l, (l+r)/2);
  build(2*p+1, (l+r)/2+1, r);
  st[p].resize(r-l+1);
  merge(st[2*p].begin(), st[2*p].end(),
    st[2*p+1].begin(), st[2*p+1].end(),
    st[p].begin());
}

int query( int p, int l, int r, int i, int j, int a, int b ) {
  if( j < l || i > r ) return 0;
  if( i <= l && j >= r )
    return upper_bound(st[p].begin(), st[p].end(), b) -
        lower_bound(st[p].begin(), st[p].end(), a);
  return query(2*p, l, (l+r)/2, i, j, a, b) +
      query(2*p+1, (l+r)/2+1, r, i, j, a, b);
}
```

4.10 Trie

```
// O(sum(|s|))
int nds = 0;
int g[MAXN][26];

void add( string s ){
  int cur = 0;
  for( char ch : s ) {
    ch -= 'a';
    if( g[cur][ch] == 0 ) g[cur][ch] = ++nds;
    cur = g[cur][ch];
  }
}

bool find( string s ) {
  int cur = 0;
  for( char ch : s ) {
    ch -= 'a';
    if( g[cur][ch] == 0 ) return false;
    cur = g[cur][ch];
  }
  return true;
}

// Bolada
struct Node {
  map<char, int> child;
  bool end;
  int getchild( char c ) {
    auto it = child.find( c );
    if( it != child.end() ) return it->second;
    return -1;
  }
};

vector<Node> trie(1);

void add( string s ) {
  int cur = 0;
  for( char c : s ) {
    if( trie[cur].getchild(c) == -1 ) {
      trie.push_back( Node() );
      trie[cur].child[c] = trie.size()-1;
    }
    cur = trie[cur].getchild(c);
  }
  trie[cur].end = true;
}

bool find( string s ) {
  int cur = 0;
  for( char c : s ) {
    if( trie[cur].getchild(c) == -1 ) return 0;
    cur = trie[cur].getchild(c);
  }
  return trie[cur].end;
}
```

4.11 Li-chao Tree

```
// Query minimum on set of functions, do not forget lc_init() before use it
// Change f() as the function changes be carefull with qudractic funcions
// O(log n) query | O(n log n) build
typedef long long ll;
typedef pair<ll, ll> pll;
inline ll f( pll a, int x ) {
  return ( a.first * x * x ) + a.second;
}

#define MAXLC 1000000
#define INF (1ll<<60)
pll line[MAXLC << 1];

void lc_init( int lo=0, int hi=MAXLC, int node=0 ) {
  if (lo > hi || line[node].second == INF) return;
  line[node] = { 0, INF };
  int mid = (lo + hi) / 2;
  lc_init( lo, mid - 1, 2 * node + 1 );
  lc_init( mid + 1, hi, 2 * node + 2 );
}

void add_line( pll ln, int lo=0, int hi=MAXLC, int node=0 ) {
  int mid = ( lo + hi ) / 2;
  bool l = f( ln, lo ) < f( line[node], lo );
  bool m = f( ln, mid ) < f( line[node], mid );
  bool h = f( ln, hi ) < f( line[node], hi );
  if( m ) swap( line[node], ln );
  if( lo == hi || ln.second == INF ) return;
  else if( l != m ) add_line( ln, lo, mid - 1, 2 * node + 1 );
  else if( h != m ) add_line( ln, mid + 1, hi, 2 * node + 2 );
}

ll get( int x, int lo=0, int hi=MAXLC, int node=0 ) {
  int mid = ( lo + hi ) / 2;
  ll ret = f( line[node], x );
  if( x < mid ) ret = min( ret, get( x, lo, mid - 1, 2 * node + 1 ) );
  if( x > mid ) ret = min( ret, get( x, mid + 1, hi, 2 * node + 2 ) );
  return ret;
}
```

4.12 Heavy Light Decomposition

```
// hld::init() to build | O( n log n ) to build and O(log n) to query/update
// Be carefull with x*10^5 limits
#define ll long long
#define MAXSEG 2*MAXN
int N;
vector<int> adj[MAXN];

namespace hld {
  int parent[MAXN];
  vector<int> ch[MAXN];
  int depth[MAXN], sz[MAXN], in[MAXN], rin[MAXN], nxt[MAXN], out[MAXN], t = 0;
  void dfs_sz( int n = 0, int p = -1, int d = 0 ) {
    parent[n] = p, sz[n] = 1, depth[n] = d;
    for( auto v : adj[n] ) if( v != p ) {
      dfs_sz( v, n, d + 1 );
      sz[n] += sz[v];
      ch[n].push_back( v );
      if( sz[v] > sz[ch[n][0]] )
        swap( ch[n][0], ch[n].back() );
    }
  }
  void dfs_hld( int n = 0 ) {
    in[n] = t++;
    rin[in[n]] = n;
    for( auto v : ch[n] ) {
      nxt[v] = ( v == ch[n][0] ? nxt[n] : v );
      dfs_hld( v );
    }
    out[n] = t;
  }

  void init() {
    dfs_sz();
    dfs_hld();
  }

  int lca( int u, int v ) {
    while( nxt[u] != nxt[v] ) {
      if( depth[nxt[u]] < depth[nxt[v]] ) swap( u, v );
      u = parent[nxt[u]];
    }
    return depth[u] < depth[v] ? u : v;
  }
  // insert segtree with lazy here
  void update_subtree( int n, int x ) {
    update( in[n], out[n] - 1, x);
  }

  ll query_subtree( int n ) {
    return query( in[n], out[n] - 1 );
  }

  void update_path( int u, int v, int x, bool ignore_lca = false ) {
    while( nxt[u] != nxt[v] ) {
      if( depth[nxt[u]] < depth[nxt[v]] ) swap( u, v );
      update( in[nxt[u]], in[u], x );
      u = parent[nxt[u]];
    }
    if( depth[u] < depth[v] ) swap( u, v );
    update( in[v] + ignore_lca, in[u], x );
  }

  ll query_path( int u, int v, bool ignore_lca = false ) {
    ll ret = 0;
    while( nxt[u] != nxt[v] ) {
      if( depth[nxt[u]] < depth[nxt[v]] ) swap( u, v );
      ret = max( ret, query( in[nxt[u]], in[u] ) );
      u = parent[nxt[u]];
    }
    if( depth[u] < depth[v] ) swap(u, v);
    ret = max( ret, query( in[v] + ignore_lca, in[u] ) );
    return ret;
  }
}
```

4.13 Link-Cut Tree

```
/*
O(1) for make_tree
O(log n) amortized for all other operations
*/
typedef long long lld;
typedef unsigned long long llu;
using namespace std;
struct Node { int L, R, P, PP, sz; };
Node LCT[MAXN];

void make_tree( int v ){
  if (v == -1) return;
  LCT[v].L = LCT[v].R = LCT[v].P = LCT[v].PP = -1;
}

void update( int v ) {
  LCT[v].sz = 1;
  if( LCT[v].L != -1 ) LCT[v].sz += LCT[LCT[v].L].sz;
  if( LCT[v].R != -1 ) LCT[v].sz += LCT[LCT[v].R].sz;
}

void rotate( int v ){
  if( v == -1 ) return;
  if( LCT[v].P == -1 ) return;
  int p = LCT[v].P;
  int g = LCT[p].P;
  if( LCT[p].L == v ) {
    LCT[p].L = LCT[v].R;
    if( LCT[v].R != -1 ) LCT[LCT[v].R].P = p;
    LCT[v].R = p;
    LCT[p].P = v;
  } else {
    LCT[p].R = LCT[v].L;
    if( LCT[v].L != -1 ) LCT[LCT[v].L].P = p;
    LCT[v].L = p;
    LCT[p].P = v;
  }
  LCT[v].P = g;
  if( g != -1 ){
    if (LCT[g].L == p) LCT[g].L = v;
    else LCT[g].R = v;
  }
  LCT[v].PP = LCT[p].PP;
  LCT[p].PP = -1;
  update( p );
}

void splay( int v ){
  if (v == -1) return;
  while( LCT[v].P != -1 ){
    int p = LCT[v].P;
    int g = LCT[p].P;
    if( g == -1 )rotate(v);
    else if( ( LCT[p].L == v ) == ( LCT[g].L == p ) ) {
      rotate( p );
```

```
      rotate( v );
    } else {
      rotate( v );
      rotate( v );
    }
  }
  update( v );
}

void expose( int v ){
  if( v == -1 ) return;
  splay( v );
  if( LCT[v].R != -1 ) {
    LCT[LCT[v].R].PP = v;
    LCT[LCT[v].R].P = -1;
    LCT[v].R = -1;
    update( v );
  }
  while( LCT[v].PP != -1 ){
    int w = LCT[v].PP;
    splay( w );
    if( LCT[w].R != -1 ) {
      LCT[LCT[w].R].PP = w;
      LCT[LCT[w].R].P = -1;
    }
    LCT[w].R = v;
    LCT[v].P = w;
    update( w );
    splay(v);
  }
}

int find_root( int v ){
  if( v == -1 ) return -1;
  expose( v );
  int ret = v;
  while( LCT[ret].L != -1 ) ret = LCT[ret].L;
  expose( ret );
  return ret;
}

void link( int v, int w ){
  if( v == -1 || w == -1 ) return;
  expose( w );
  LCT[v].L = w;
  LCT[w].P = v;
  LCT[w].PP = -1;
  update( v );
}

int depth( int v ) {
  expose( v );
  return LCT[v].sz - 1;
}

void cut( int v ){
  if( v == -1 ) return;
  expose( v );
  if( LCT[v].L != -1 ){
    LCT[LCT[v].L].P = -1;
    LCT[LCT[v].L].PP = -1;
    LCT[v].L = -1;
  }
  update( v );
}

bool connected( int p, int q) {
  return find_root( p ) == find_root( q );
}

int LCA( int p, int q ){
  expose( p );
  splay( q );
  if( LCT[q].R != -1 ) {
    LCT[LCT[q].R].PP = q;
    LCT[LCT[q].R].P = -1;
    LCT[q].R = -1;
  }
  int ret = q, t = q;
  while( LCT[t].PP != -1 ) {
    int w = LCT[t].PP;
    splay( w );
    if( LCT[w].PP == -1 ) ret = w;
    if( LCT[w].R != -1 ) {
      LCT[LCT[w].R].PP = w;
      LCT[LCT[w].R].P = -1;
    }
    LCT[w].R = t;
    LCT[t].P = w;
    LCT[t].PP = -1;
    t = w;
  }
  splay( q );
  return ret;
}
```

4.14 Mo's algorithm (sqrt decomp)

```
// Square Root Decomposition (Mo's Algorithm) - O(n^(3/2))
// SQ is in this proportion: 10^5 -> 500
int n, m, v[MAXN];

void add(int p) { /* add value to aggregated data structure */ }
void rem(int p) { /* remove value from aggregated data structure */ }

struct query { int i, l, r, ans; } qs[MAXN];

bool c1( query a, query b ) {
  if(a.l/SQ != b.l/SQ) return a.l < b.l;
  return a.l/SQ&1 ? a.r > b.r : a.r < b.r;
}

bool c2( query a, query b ) { return a.i < b.i; }

/* inside main */
int l = 0, r = -1;
sort( qs, qs+m, c1 );
for (int i = 0; i < m; ++i) {
  query &q = qs[i];
  while (r < q.r) add(v[++r]);
  while (r > q.r) rem(v[r--]);
  while (l < q.l) rem(v[l++]);
  while (l > q.l) add(v[--l]);
  q.ans = /* calculate answer */;
}

sort(qs, qs+m, c2); // sort to original order
```

5 Strings

5.1 Aho Corasick Automata

```
// Aho Corasick automaton O(N + sum(|S|)) / m is the number of states in
     automaton
#define ll long long
#define OFF 'a'
#define MAX_N 100013
int n; // size of dictionary
string dict[MAX_N];
string text;

#define MAX_M 100013
int g[MAX_M][26]; // the normal edges in the trie
int f[MAX_M]; // failure function
ll out[MAX_M]; // output function

int aho_corasick() {
  memset( g, -1, sizeof g );
  memset( out, 0, sizeof out );
  int nodes = 1;
  for( int i = 0 ; i < n ; ++i ) {
    string& s = dict[i];
    int cur = 0;

    for( int j = 0; j < s.size(); ++j ) {
      if (g[cur][s[j] - OFF] == -1) g[cur][s[j] - OFF] = nodes++;
      cur = g[cur][s[j] - OFF];
    }
    ++out[cur];
  }

  for( int ch = 0 ; ch < 26 ; ++ch ) if( g[0][ch] == -1 ) g[0][ch] = 0;

  memset( f, -1, sizeof f );
  queue<int> q;
  for( int ch = 0 ; ch < 26 ; ++ch ) {
    if( g[0][ch] != 0 ) {
      f[g[0][ch]] = 0;
      q.push( g[0][ch] );
    }
  }

  while( !q.empty() ) {
    int state = q.front();
    q.pop();

    for( int ch = 0 ; ch < 26 ; ++ch ) {
      if( g[state][ch] == -1 ) continue;

      int fail = f[state];
      while( g[fail][ch] == -1 ) fail = f[fail];

      f[g[state][ch]] = g[fail][ch];
      out[g[state][ch]] += out[g[fail][ch]];

      q.push( g[state][ch] );
    }
  }

  return nodes;
}

ll search() {
  int state = 0;
  ll ret = 0;
  for( char c : text ) {
    while( g[state][c - OFF] == -1 ) state = f[state];
    state = g[state][c - OFF];
    ret += out[state];
  }
  return ret;
}
```

5.2 Z pattern search

```
// Z[i] stores length of the longest substring starting from st[i]
// which is also prefix of str[0..n-1].
// O(|P|+|S|)
int Z[MAXN], m[MAXN];

void z_do( string S ) {
  int N = S.size(), L = 0, R = 0;
  Z[0] = N;
  for( int i = 1 ; i < N ; ++i ) {
    if( i < R ) Z[i] = min( R - i, Z[i - L] );
    while( i + Z[i] < N && S[i + Z[i]] == S[Z[i]] ) ++Z[i];
    if( i + Z[i] > R ) L = i, R = i + Z[i];
  }
}

int search( string S, string P ) {
  int N = S.size(), M = P.size(), msize = 0;
  string combined = P + S;
  z_do( combined );
  for( int i = 0 ; i < N ; ++i )
    if( Z[M + i] >= M ) m[msize++] = i;
  return msize;
}
```

5.3 KMP

```
//Pattern search O(|T|+|P|)
vector<int> comp_shifts(string P) {
  int p = P.length();
  vector<int> shifts(p);
  for (int q = 1; q < p; q++) {
    int k = shifts[q - 1];
    while (k > 0 && P[k] != P[q])
      k = shifts[k - 1];
    if (P[k] == P[q])
      k++;
    shifts[q] = k;
  }
  return shifts;
}

int kmp(string P, string T) {
  vector<int> shifts = comp_shifts(P);
  int n = T.length();
  int m = P.length();

  int occurrences = 0;
  int q = 0;
  for (int i = 0; i < n; i++) {
    while (q && P[q] != T[i])
      q = shifts[q - 1];
    if (P[q] == T[i])
      q++;
    if (q == m) {
      occurrences++;
      q = shifts[q - 1];
    }
  }
  return occurrences;
}
```

5.4 Hashing pattern

```
// Rabin-karp O(n+m)
const int B = 31;
char s[MAXN], p[MAXN];
int n, m; // n = strlen(s), m = strlen(p)

void rabin() {
  if( n<m ) return;
  ull hp = 0, hs = 0, E = 1;
  for (int i = 0; i < m; ++i)
    hp = ((hp*B)%MOD + p[i])%MOD,
    hs = ((hs*B)%MOD + s[i])%MOD,
    E = (E*B)%MOD;

  if (hs == hp) { /* matching position 0 */ }
  for( int i = m ; i < n ; ++i ) {
    hs = ((hs*B)%MOD + s[i])%MOD;
    hhs = (hs - s[i-m]*E%MOD + MOD)%MOD;
    if( hs == hp ) { /* matching position i-m+1 */ }
  }
}

// Good hashing :) O(n+m)
typedef long long LL;
typedef pair<LL, LL> pll;

const int MOD = 1e9 + 7;
const pll BASE = {4441, 7817};

pll operator+(const pll& a, const pll& b) {
  return { (a.first + b.first) % MOD, (a.second + b.second) % MOD };
}
pll operator+(const pll& a, const LL& b) {
  return { (a.first + b) % MOD, (a.second + b) % MOD };
}
pll operator-(const pll& a, const pll& b) {
  return { (MOD + a.first - b.first) % MOD, (MOD + a.second - b.second) % MOD };
}
pll operator*(const pll& a, const pll& b) {
  return { (a.first * b.first) % MOD, (a.second * b.second) % MOD };
}
pll operator*(const pll& a, const LL& b) {
  return { (a.first * b) % MOD, (a.second * b) % MOD };
}

pll get_hash(string s) {
  pll h = {0, 0};
  for (int i = 0; i < s.size(); i++) {
    h = BASE * h + s[i];
  }
  return h;
}

struct hsh {
  int N;
  string S;
  vector<pll> pre, pp;

  void init(string S_) {
    S = S_;
    N = S.size();
    pp.resize(N);
    pre.resize(N + 1);
    pp[0] = {1, 1};
    for (int i = 0; i < N; i++) {
      pre[i + 1] = pre[i] * BASE + S[i];
      if (i) { pp[i] = pp[i - 1] * BASE; }
    }
  }

  pll get(int s, int e) {
    return pre[e] - pre[s] * pp[e - s];
  }
};

vector<int> search(string s, string p) {
  vector<int> matches;
  pll h = get_hash(p);
  hsh hs; hs.init(s);
  for (int i = 0; i + p.size() <= s.size(); i++) {
    if (hs.get(i, i + p.size()) == h) {
      matches.push_back(i);
    }
  }
  return matches;
}
```

5.5 Suffix Array + LCP

```
// O(n log(n) )
vector<int> suffix_array( string S ) {
  int N = S.size();
  vector<int> sa( N ), classes( N );
  for( int i = 0 ; i < N ; ++i ) sa[i] = N - 1 - i, classes[i] = S[i];
  stable_sort( sa.begin(), sa.end(), [&S]( int i, int j ) {
    return S[i] < S[j];
  } );
  for( int len = 1 ; len < N ; len *= 2 ) {
    vector<int> c( classes );
    for( int i = 0; i < N; ++i ) {
      bool same = i && sa[i - 1] + len < N
                  && c[sa[i]] == c[sa[i - 1]]
                  && c[sa[i] + len / 2] == c[sa[i - 1] + len / 2];
      classes[sa[i]] = same ? classes[sa[i - 1]] : i;
    }
    vector<int> cnt( N ), s( sa );
    for( int i = 0 ; i < N ; ++i ) cnt[i] = i;
    for( int i = 0 ; i < N ; ++i ) {
      int s1 = s[i] - len;
      if( s1 >= 0 )
        sa[cnt[classes[s1]]++] = s1;
    }
  }
  return sa;
}

vector<int> LCP( const vector<int>& sa, string S ) {
  int N = S.size();
  vector<int> rank( N ), lcp( N - 1 );
  for( int i = 0 ; i < N ; ++i ) rank[sa[i]] = i;
  int pre = 0;
  for( int i = 0 ; i < N ; ++i ) {
    if( rank[i] < N - 1 ) {
      int j = sa[rank[i] + 1];
      while( max( i, j ) + pre < S.size() && S[i + pre] == S[j + pre] ) ++pre;
      lcp[rank[i]] = pre;
      if( pre > 0 ) --pre;
    }
  }
  return lcp;
}

// Longest Repeated Substring O(n)
```

```cpp
int lrs = 0;
for( int i = 0 ; i < n ; ++i ) lrs = max(lrs, lcp[i]);

// Longest Common Substring O(n)
// m = strlen(s);
// strcat(s, "$"); strcat(s, p); strcat(s, "#");
// n = strlen(s);
int lcs = 0;
for( int i = 1 ; i < n ; ++i ) if ( ( sa[i] < m ) != ( sa[i - 1] < m ) )
  lcs = max(lcs, lcp[i]);

// To calc LCS for multiple texts use a slide window with minqueue
// The number of different substrings of a string is n*(n + 1)/2 - sum(lcs[i])
```

5.6 Longest palindromic string

```cpp
// d1, d2 = number of palindromes with odd and even lengths with centers in i
vector<int> d1, d2;

void manacher( string s ){
  int n = s.length();
  // odd
  d1.resize(n);
  for (int i = 0, l = 0, r = -1; i < n; i++) {
    int k = (i > r) ? 1 : min(d1[l + r - i], r - i + 1);
    while (0 <= i - k && i + k < n && s[i - k] == s[i + k]) k++;
    d1[i] = k--;
    if (i + k > r) l = i - k, r = i + k;
  }
  // even
  d2.resize(n);
  for (int i = 0, l = 0, r = -1; i < n; i++) {
    int k = (i > r) ? 0 : min(d2[l + r - i + 1], r - i + 1);
    while (0 <= i - k - 1 && i + k < n && s[i - k - 1] == s[i + k]) k++;
    d2[i] = k--;
    if (i + k > r) l = i - k - 1, r = i + k ;
  }
}

// To get the string just str.substr( ( id + 1 - mx ) / 2, mx ) | mx is the size
     of the LPS
// O(n)
pair<int, int> manacher( string str ){
  int i, j, k, l = str.length(), n = l << 1, mx = -1, id;
  vector<int> pal( n );
  for( i = 0, j = 0, k = 0 ; i < n ; j = max( 0, j - k ), i += k ) {
    while( j <= i && ( i + j + 1 ) < n && str[( i - j ) >> 1] == str[( i + j + 1
          ) >> 1] ) ++j;
    for( k = 1, pal[i] = j; k <= i && k <= pal[i] && ( pal[i] - k ) != pal[i - k
         ]; ++k )
      pal[i + k] = min( pal[i - k], pal[i] - k );
    if( pal[i] > mx ) mx = pal[i], id = i;
  }
  pal.pop_back();
  return { mx, id };
}
```

5.7 Suffix automaton

```cpp
// Suffix Automaton Construction - O(n) FROM IME

const int N = 1e6+1, K = 26;
int sl[2*N], len[2*N], sz, last;
ll cnt[2*N];
map<int, int> adj[2*N];

void add(int c) {
  int u = sz++;
  len[u] = len[last] + 1;
  cnt[u] = 1;

  int p = last;
  while(p != -1 and !adj[p][c])
    adj[p][c] = u, p = sl[p];

  if (p == -1) sl[u] = 0;
  else {
    int q = adj[p][c];
    if (len[p] + 1 == len[q]) sl[u] = q;
    else {
      int r = sz++;
      len[r] = len[p] + 1;
      sl[r] = sl[q];
      adj[r] = adj[q];
      while(p != -1 and adj[p][c] == q)
        adj[p][c] = r, p = sl[p];
      sl[q] = sl[u] = r;
    }
  }

  last = u;
}

void clear() {
  for(int i=0; i<=sz; ++i) adj[i].clear();
  last = 0;
  sz = 1;
  sl[0] = -1;
}

void build(char *s) {
  clear();
  for(int i=0; s[i]; ++i) add(s[i]);
}

// Pattern matching - O(|p|)
bool check(char *p) {
  int u = 0, ok = 1;
  for(int i=0; p[i]; ++i) {
    u = adj[u][p[i]];
    if (!u) ok = 0;
  }
  return ok;
}

// Substring count - O(|p|)
ll d[2*N];

void substr_cnt(int u) {
  d[u] = 1;
  for(auto p : adj[u]) {
    int v = p.second;
    if (!d[v]) substr_cnt(v);
    d[u] += d[v];
  }
}

ll substr_cnt() {
  memset(d, 0, sizeof d);
  substr_cnt(0);
  return d[0] - 1;
}
```

```
// k-th Substring - O(|s|)
// Just find the k-th path in the automaton.
// Can be done with the value d calculated in previous problem.

// Smallest cyclic shift - O(|s|)
// Build the automaton for string s + s. And adapt previous dp
// to only count paths with size |s|.


// Number of occurences - O(|p|)
vector<int> t[2*N];

void occur_count(int u) {
  for(int v : t[u]) occur_count(v), cnt[u] += cnt[v];
}

void build_tree() {
  for(int i=1; i<=sz; ++i)
    t[sl[i]].push_back(i);
  occur_count(0);
}

ll occur_count(char *p) {
  // Call build tree once per automaton
  int u = 0;
  for(int i=0; p[i]; ++i) {
    u = adj[u][p[i]];
    if (!u) break;
  }
  return !u ? 0 : cnt[u];
}

// First occurence - (|p|)
// Store the first position of occurence fp.
// Add the the code to add function:
// fp[u] = len[u] - 1;
// fp[r] = fp[q];

// To answer a query, just output fp[u] - strlen(p) + 1
// where u is the state corresponding to string p

// All occurences - O(|p| + |ans|)
// All the occurences can reach the first occurence via suffix links.
// So every state that contains a occurence is reacheable by the
// first occurence state in the suffix link tree. Just do a DFS in this
// tree, starting from the first occurence.
// OBS: cloned nodes will output same answer twice.


// Smallest substring not contained in the string - O(|s| * K)
// Just do a dynamic programming:
// d[u] = 1 // if d does not have 1 transition
// d[u] = 1 + min d[v] // otherwise


// LCS of 2 Strings - O(|s| + |t|)
// Build automaton of s and traverse the automaton wih string t
// mantaining the current state and the current lenght.
// When we have a transition: update state, increase lenght by one.
// If we don't update state by suffix link and the new lenght will
// should be reduced (if bigger) to the new state length.
// Answer will be the maximum length of the whole traversal.

// LCS of n Strings - O(n*|s|*K)
// Create a new string S = s_1 + d1 + ... + s_n + d_n,
// where d_i are delimiters that are unique (d_i != d_j).
// For each state use DP + bitmask to calculate if it can
// reach a d_i transition without going through other d_j.
// The answer will be the biggest len[u] that can reach all
// d_i's.
```

5.8 Palindromic Tree

```
// usage, cin >> s; foreach i -> len(s) : insert(i)
// lps = longest palindromic substring
// num = number of palindromes in substring
// ptr-2 = number of different palindromic substrings
struct Node {
  int start, end;
  int len;
  int num;
  // change to map if both cases (watch for TLE)
  int next[27];
  int link;
};

Node tree[MAXN];
int currNode;
int lps;
string s;
int ptr;

void insert(int idx) {
  int tmp = currNode;
  int let = s[idx] - 'a';// Watch!!
  while(!(idx - tree[tmp].len >= 1 && s[idx] == s[idx-tree[tmp].len-1]))
    tmp = tree[tmp].link;
  if(tree[tmp].next[let] != 0) {
    currNode = tree[tmp].next[let];
    return;
  }
  ptr++;
  tree[tmp].next[let] = ptr;
  tree[ptr].len = tree[tmp].len + 2;
  tree[ptr].end = idx;
  tree[ptr].start = idx - tree[ptr].len + 1;
  tmp = tree[tmp].link;
  currNode = ptr;
  lps = max( lps, tree[ptr].len );
  if(tree[currNode].len == 1) {
    tree[currNode].link = 2;
    tree[currNode].num = 1;
    return;
  }
  while(!(idx-tree[tmp].len >= 1 && s[idx] == s[idx-tree[tmp].len-1]))
    tmp = tree[tmp].link;
  tree[currNode].link = tree[tmp].next[let];
  tree[currNode].num = 1 + tree[tree[currNode].link].num;
}

void init() {
  tree[1].len = -1;
  tree[1].link = tree[2].link = 1;
  tree[2].len = 0;
  ptr = 2, currNode = 1;
}
```

6 Dynamic programming

6.1 Knapsack problems

```
// knapsack 0-1 O(n * wei) | index 0
// maximum profit for weight j
// wei is max weigth
// v is price, w is weight dp[MAXWEIGHT+1]
for( int i = 0 ; i < n ; ++i )
  for( int j = wei ; j >= w[i] ; --j )
    dp[j] = max( dp[j], v[i] + dp[j - w[i]] );

// repetition allowed with items dp[0] is pred dp[1] is formula
// bb is max weight, n is size
// wei = weights, val = values
for( int i = 0 ; i <= bb ; ++i ) {
  for( int j = 0 ; j < n ; ++j ) {
    if( i >= wei[j] ){
      dp[1][i] = max( dp[1][i], val[j] + dp[1][i - wei[j]] );
      dp[0][i] = j;
    }
  }
}
int m = bb;
while( m != 0 ){
  // access weight with wei[dp[0][m]]
  m -= wei[dp[0][m]];
}

// knapsack
// F[a] := minimum weight for profit a
int knapsackP(vector<int> p, vector<int> w, int c) {
  int n = p.size(), P = accumulate(p.begin(), p.end(), 0);
  vector<int> F(P+1, c+1); F[0] = 0;
  for (int i = 0; i < n; ++i)
    for (int a = P; a >= p[i]; --a)
      F[a] = min(F[a], F[a-p[i]] + w[i]);
  for (int a = P; a >= 0; --a) if (F[a] <= c) return a;
}

// knapsack with itens in order
val[n] = 0;
reverse(val, val+n+1);
for( int i = 1 ; i <= n ; ++i ) {
  for( int j = wei ; j >= val[i] ; --j ) {
    if( dp[i-1][j] > dp[i-1][j-val[i]]+val[i] )
      dp[i][j] = dp[i-1][j];
    else
      dp[i][j] = dp[i-1][j-val[i]] + val[i],
      dp2[i][j] = 1;
  }
  for( int j = val[i] - 1 ; j >= 0 ; --j ) dp[i][j] = dp[i-1][j];
}
int k = wei;
for( int i = n ; i > 0 ; --i )
  if( dp2[i][k] ) printf("%d ", val[i] ), k -= val[i];
printf("%d\n", dp[n][wei] );

// bounded knapsack
// ps = values ; ws = weights
// ms = quantity ; W = weight wanted ; n = item quantity
int solve(){
  int dp[n+1][W+1];
  for( int i = 0; i < n; ++i ) {
    for( int s = 0; s < ws[i]; ++s ) {
      int alpha = 0;
      queue<int> que;
      deque<int> peek;
      for( int w = s ; w <= W ; w += ws[i] ) {
        alpha += ps[i];
        int a = dp[i][w]-alpha;
        que.push( a );
        while( !peek.empty() && peek.back() < a ) peek.pop_back();
        peek.push_back(a);
        while( que.size() > ms[i]+1 ) {
          if (que.front() == peek.front()) peek.pop_front();
          que.pop();
        }
        dp[i+1][w] = peek.front()+alpha;
      }
    }
  }
  int ans = 0;
  for( int w = 0 ; w <= W ; ++w )
    ans = max( ans, dp[n][w] );
  return ans;
}

// Branch and bound, O(2^c) where c is small most of time
template <class T>
struct knapsack {
  T c;
  struct item { T p, w; };
  vector<item> is;
  void add_item(T p, T w) {
    is.push_back({p, w});
  }
  T det(T a, T b, T c, T d) {
    return a * d - b * c;
  }
  T z;
  void expbranch(T p, T w, int s, int t) {
    if (w <= c) {
      if (p >= z) z = p;
      for (; t < is.size(); ++t) {
        if (det(p - z - 1, w - c, is[t].p, is[t].w) < 0) return;
        expbranch(p + is[t].p, w + is[t].w, s, t + 1);
      }
    } else {
      for (; s >= 0; --s) {
        if (det(p - z - 1, w - c, is[s].p, is[s].w) < 0) return;
        expbranch(p - is[s].p, w - is[s].w, s - 1, t);
      }
    }
  }
  T solve() {
    sort(is.begin(), is.end(), [](const item &a, const item &b) {
      return a.p * b.w > a.w * b.p;
    });
    T p = 0, w = 0;
    z = 0;
    int b = 0;
    for (; b < is.size() && w <= c; ++b) {
      p += is[b].p;
      w += is[b].w;
    }
    expbranch(p, w, b-1, b);
    return z;
  }
};
```

6.2 Coin problems

```
//subset sum O(n*sum)
dp[0] = 1;
for( int i = 0 ; i < n ; ++i )
  for(int j = sum ; j >= v[i] ; --j )
    dp[j] |= dp[j-v[i]];

// bitset optimization O(n*sum/(32|64))
```

```
bitset<MAXSUM> dp;
dp.set( 0 );
for( int i = 0 ; i < n ; ++i )
  dp |= dp << v[i];

// coin change
#define INF 0x3f3f3f3f
// find the minimum number of coin changes
// coins = vector with values, n is size
int coin_change( int amt ){
  int dp[amt+1];
  int pred[amt+1];
  for( int i = 0 ; i <= amt ; ++i ) pred[i] = 0, dp[i] = INF;
  dp[0] = 0;
  for( int i = 1 ; i <= amt ; ++i ){
    int mini = dp[i];
    for (int j = 0 ; j < n ; ++j ){
      if( i >= coins[j] ){
        mini = min( mini, dp[i-coins[j]] + 1 );
        pred[i] = j;
      }
    }
    dp[i] = mini;
  }
  // get each coin used
  int m = amt;
  while( m != 0 ){
    //process here, coin value at coins[pred[m]]
    m -= coins[pred[m]];
  }
  return dp[amt];
}
```

6.3 Longest Zigzag

```
// A sequence xs is zigzag if x[i] < x[i+1], x[i+1] > x[i+2], for all i
// (initial direction can be arbitrary). The maximum length zigzag
// subsequence is computed in O(n) time by a greedy method.
int longestZigZagSubsequence( vector<int> xs ) {
  int n = xs.size(), len = 1, prev = -1;
  for( int i = 0, j; i < n; i = j ){
    for( j = i+1 ; j < n && xs[i] == xs[j] ; ++j );
    if (j < n) {
      int sign = (xs[i] < xs[j]);
      if (prev != sign) ++len;
      prev = sign;
    }
  }
  return len;
}
int longestZigZagSubsequence(vector<int> A) {
  int n = A.size();
  int Z[n][2];
  Z[0][0] = 1;
  Z[0][1] = 1;
  int best = 1;
  for( int i = 1; i < n; ++i ){
    for( int j = i-1; j>= 0; --j ){
      if( A[j] < A[i] ) Z[i][0] = max( Z[j][1]+1, Z[i][0] );
      if( A[j] > A[i] ) Z[i][1] = max( Z[j][0]+1, Z[i][1] );
    }
    best = max( best, max( Z[i][0], Z[i][1] ) );
  }
  return best;
}
```

6.4 DP on Trees

```
// Count sub tree
// dp[u][j] = # of different sub trees of size less than or equal to K.
// g[i] is childrens of i
vector<int> g[MAXN];
int dp[MAXN][MAXK], sub[MAXN], tmp[MAXK];
int k;
void dfs( int u ) {
  sub[u] = 1;
  dp[u][0] = dp[u][1] = 1;
  for( int v : g[u] ) {
    dfs( v );
    fill( tmp , tmp + k + 1 , 0 );
    for( int i = 1 ; i <= min( sub[u] , k ) ; ++i )
      for( int j = 0 ; j <= sub[v] && i + j <= k ; ++j )
        tmp[i + j] += dp[u][i] * dp[v][j];
    sub[u] += sub[v];
    for( int i = 0 ; i <= min( k , sub[u] ) ; ++i )
      dp[u][i] = tmp[i];
  }
}

//Longest path on DAG O(n+m), index 1

int dp[MAXN];

void dfs( int u ) {
  vis[u] = true;
  for( int v : g[u] ) {
    if( !vis[v] ) dfs( v );
    dp[u] = max( dp[u], 1+ dp[v] );
  }
}

int lp() {
  for( int i = 1 ; i <= n ; ++i ) if( !vis[i] ) dfs( i );
  int r = 0;
  for( int i = 1 ; i <= n ; ++i ) r = max( r, dp[i] );
  return r;
}
```

6.5 Longest Increasing Subsequence

```
// O(n log n)
vector<int> lis( vector<int> v ) {
  vector<pair<int, int> > best;
  vector<int> dad( v.size(), -1 );
  for( int i = 0 ; i < v.size() ; ++i ) {
    pair<int, int> item = make_pair( v[i], 0 );
    auto it = lower_bound( best.begin(), best.end(), item );
    item.second = i;
    /* non-decreasing
    pair<int, int> item = make_pair(v[i], i);
    auto it = upper_bound( best.begin(), best.end(), item );
    */
    if( it == best.end() ) {
      dad[i] = ( best.size() == 0 ? -1 : best.back().second );
      best.push_back( item );
    } else {
      dad[i] = it == best.begin() ? -1 : prev( it )->second;
      *it = item;
    }
  }
```

```
  vector<int> ret;
  for( int i = best.back().second ; i >= 0 ; i = dad[i] ) ret.push_back( v[i] );
  reverse( ret.begin(), ret.end() );
  return ret;
}

// Only size of lis
int lis( vector<int> v ) {
  int dp[v.size() + 10], lis = -1;
  memset( dp, 0x3f, sizeof dp );
  for( int i : v ) {
    int j = lower_bound( dp, dp + lis, i ) - dp;
    dp[j] = min( dp[j], i );
    lis = max( lis, j + 1 );
  }
  return lis;
}


// lis O(n^2) and count how many lises are, please take care of long long
// dp[i] stores length of the lis ending at i
// tot[i] stores how many ways we can obtain the lis ending in the values d[i]

int tot[MAXN];
int dp[MAXN];

pair<int, int> lis( vector<int> a ) {
  int lis = 1;
  for( int i = 0 ; i < a.size() ; ++i ) {
    dp[i] = 1;
    tot[i] = 1;
    for( int j = 0 ; j < i ; ++j ) {
      if( a[j] < a[i] ) {
        if( dp[i] < dp[j] + 1 ) {
          dp[i] = dp[j] + 1;
          tot[i] = tot[j];
          lis = max( lis, dp[i] );
        } else if( dp[i] == dp[j] + 1 ) {
          tot[i] = ( tot[i] + tot[j] ) % MOD;
        }
      }
    }
  }
  int qnt = 0;
  for( int i = 0 ; i < a.size() ; ++i ) {
    if( dp[i] == lis ) {
      qnt = ( qnt + tot[i] ) % MOD;
    }
  }
  return {lis, qnt};
}
```

6.6 Longest Common Subsequence

```
// O(m * n)
// to compute only size use:
int lcs( string &X, string &Y ) {
  int m = X.length(), n = Y.length();
  int L[2][n + 1];
  bool bi;
  for( int i = 0 ; i <= m ; ++i ) {
    bi = i & 1;
    for( int j = 0; j <= n ; ++j ) {
      if (i == 0 || j == 0)  L[bi][j] = 0;
      else if( X[i-1] == Y[j-1] ) L[bi][j] = L[1 - bi][j - 1] + 1;
      else L[bi][j] = max(L[1 - bi][j], L[bi][j - 1]);
    }
  }
  return L[bi][n];
}
//to compute sequence:
typedef vector<int> vi;
typedef vector<vi> vvi;

void backtrack( vvi &dp, vi &res, vi &A, vi &B, int i, int j ) {
  if( !i || !j ) return;
  if( A[i-1] == B[j-1] )
    res.push_back( A[i-1] ), backtrack( dp, res, A, B, i - 1, j - 1 );
  else
    if( dp[i][j-1] >= dp[i-1][j] ) backtrack( dp, res, A, B, i, j - 1 );
    else backtrack( dp, res, A, B, i - 1, j );
}

void backtrackall( vvi &dp, set<vi> &res, vi &A, vi &B, int i, int j ) {
  if( !i || !j ) { res.insert(vi()); return; }
  if( A[i-1] == B[j-1] ) {
    set<vi> tempres;
    backtrackall( dp, tempres, A, B, i - 1, j - 1 );
    for( auto it = tempres.begin() ; it!=tempres.end() ; ++it ) {
      vi temp = *it;
      temp.push_back( A[i-1] );
      res.insert( temp );
    }
  }
  else
  {
    if( dp[i][j-1] >= dp[i-1][j] ) backtrackall( dp, res, A, B, i, j - 1 );
    if( dp[i][j-1] <= dp[i-1][j] ) backtrackall( dp, res, A, B, i - 1, j );
  }
}

vi LCS( vi &A, vi &B ) {
  vvi dp;
  int n = A.size(), m = B.size();
  dp.resize( n + 1 );
  for( int i = 0 ; i <= n ; ++i ) dp[i].resize( m + 1, 0 );
  for( int i = 1 ; i <= n ; ++i )
    for( int j = 1 ; j <= m ; ++j )
      if( A[i-1] == B[j-1] ) dp[i][j] = dp[i-1][j-1]+1;
      else dp[i][j] = max( dp[i-1][j], dp[i][j-1] );
  vi res;
  backtrack( dp, res, A, B, n, m );
  reverse( res.begin(), res.end() );
  return res;
}

set<vi> LCSall( vi &A, vi &B ) {
  vvi dp;
  int n = A.size(), m = B.size();
  dp.resize( n + 1 );
  for( int i = 0 ; i <= n ; ++i ) dp[i].resize(m+1, 0);
  for( int i = 1 ; i <= n ; ++i )
    for(int j = 1 ; j <= m ; ++j )
      if( A[i-1] == B[j-1] ) dp[i][j] = dp[i-1][j-1]+1;
      else dp[i][j] = max( dp[i-1][j], dp[i][j-1] );
  set<vi> res;
  backtrackall( dp, res, A, B, n, m );
  return res;
}
```

6.7 Convex hull trick

```
//O(n log n )
#define ll long long
```

```
struct Point{
  ll x, y;
  Point( ll x = 0, ll y = 0 ) : x(x), y(y) {}
  Point operator-( Point p ){ return Point(x - p.x, y - p.y); }
  Point operator+( Point p ){ return Point(x + p.x, y + p.y); }
  Point ccw(){ return Point( -y, x ); }
  ll operator%( Point p ){ return x*p.y - y*p.x; }
  ll operator*( Point p ){ return x*p.x + y*p.y; }
  bool operator<( Point p ) const { return x == p.x ? y < p.y : x < p.x; }
};

pair<vector<Point>, vector<Point>> ch( Point *v ) {
  vector<Point> hull, vecs;
  for( int i = 0; i < n; ++i ) {
    if( hull.size() and hull.back().x == v[i].x ) continue;
    while( vecs.size() and vecs.back()*( v[i] - hull.back() ) <= 0 )
      vecs.pop_back(), hull.pop_back();
    if( hull.size() )
      vecs.pb( ( v[i] - hull.back() ).ccw() );
    hull.pb( v[i] );
  }
  return { hull, vecs };
}

ll get(ll x) {
    Point query = {x, 1};
    auto it = lower_bound(vecs.begin(), vecs.end(), query, [](Point a, Point b)
        {
        return a%b > 0;
    });
    return query*hull[it - vecs.begin()];
}
```

6.8 Knuth Optimization

```
// Knuth DP Optimization - O(n^3) -> O(n^2) from IME
//
// 1) dp[i][j] = min i<k<j { dp[i][k] + dp[k][j] } + C[i][j]
// 2) dp[i][j] = min k<i { dp[k][j-1] + C[k][i] }
//
// Condition: A[i][j-1] <= A[i][j] <= A[i+1][j]
// A[i][j] is the smallest k that gives an optimal answer to dp[i][j]
int n;
int dp[MAXN][MAXN], a[MAXN][MAXN];
int cost( int i, int j ) {
}

void knuth() {
  // calculate base cases
  memset( dp, 63, sizeof( dp ) );
  for( int i = 1 ; i <= n ; ++i ) dp[i][i] = 0;

  // set initial a[i][j]
  for( int i = 1 ; i <= n ; ++i ) a[i][i] = i;

  for( int j = 2 ; j <= n ; ++j )
    for( int i = j ; i >= 1 ; --i )
      for( int k = a[i][j-1]; k <= a[i+1][j]; ++k ) {
        ll v = dp[i][k] + dp[k][j] + cost(i, j);
        // store the minimum answer for d[i][k]
        // in case of maximum, use v > dp[i][k]
        if (v < dp[i][j])
          a[i][j] = k, dp[i][j] = v;
      }
}
```

```
// 2) dp[i][j] = min k<i { dp[k][j-1] + C[k][i] }
int n, maxj;
int dp[N][J], a[N][J];

// declare the cost function
int cost( int i, int j ) {
  // ...
}

void knuth() {
  // calculate base cases
  memset( dp, 63, sizeof( dp ) );
  for ( int i = 1 ; i <= n ; ++i ) dp[i][1] = // ...

  // set initial a[i][j]
  for( int i = 1 ; i <= n ; ++i ) a[i][0] = 0, a[n+1][i] = n;

  for( int j = 2 ; j <= maxj ; ++j )
    for( int i = n ; i >= 1 ; --i )
      for( int k = a[i][j-1]; k <= a[i+1][j]; ++k ) {
        ll v = dp[k][j-1] + cost(k, i);
        // store the minimum answer for d[i][k]
        // in case of maximum, use v > dp[i][k]
        if( v < dp[i][j] )
          a[i][j] = k, dp[i][j] = v;
      }
}
```

6.9 Divide and conquer Optimization

```
// Divide and Conquer DP Optimization - O(k*n^2) => O(k*n*logn) FROM IME
//
// dp[i][j] = min k<i { dp[k][j-1] + C[k][i] }
//
// Condition: A[i][j] <= A[i+1][j]
// A[i][j] is the smallest k that gives an optimal answer to dp[i][j]
int n, maxj;
int dp[MAXN][MAXM], a[MAXN][MAXM];

// declare the cost function
int cost( int i, int j ) {
  // ...
}

void calc( int l, int r, int j, int kmin, int kmax ) {
  int m = ( l + r )/2;
  dp[m][j] = LINF;
  for( int k = kmin; k <= kmax; ++k ) {
    ll v = dp[k][j-1] + cost( k, m );
    // store the minimum answer for d[m][j]
    // in case of maximum, use v > dp[m][j]
    if( v < dp[m][j] ) a[m][j] = k, dp[m][j] = v;
  }

  if( l < r ) {
    calc( l, m, j, kmin, a[m][k] );
    calc( m + 1, r, j, a[m][k], kmax );
  }
}

// run for every j
for( int j = 2; j <= maxj; ++j )
  calc( 1, n, j, 1, n );
```

6.10 Digit DP

```
// framework to solve problems of counting the numbers less (O(n))
// than equal to given number whose digits satisfy constraint
// it computes
//      sum { prod(x) : 0 <= x <= z }
//    where
//      prod(x) = (((e * x[0]) * x[1])...) * x[n-1].
// struct Value {
//   Value &operator+(Value y)
//   Value &operator*(int d)
// };
// struct Automaton {
//   int init
//   int size()
//   int next(int state, int d)
//   bool accept(int state)
// };
template <class Value, class Automaton>
Value digitDP(string z, Value e, Automaton M, bool eq = 1) {
  struct Maybe {
    Value value;
    bool undefined = true;
  };
  auto oplusTo = [&](Maybe &x, Maybe y) {
    if (x.undefined) x = y;
    else if (!y.undefined) x.value += y.value;
  };
  auto otimes = [&](Maybe x, int d) {
    x.value *= d;
    return x;
  };
  int n = z.size();
  vector<vector<Maybe>> curr(2, vector<Maybe>(M.size()));
  curr[1][M.init] = {e, false};
  for (int i = 0; i < n; ++i) {
    vector<vector<Maybe>> next(2, vector<Maybe>(M.size()));
    for (int tight = 0; tight <= 1; ++tight) {
      for (int state = 0; state < M.size(); ++state) {
        if (curr[tight][state].undefined) continue;
        int lim = (tight ? z[i] - '0' : 9);
        for (int d = 0; d <= lim; ++d) {
          int tight_ = tight && d == lim;
          int state_ = M.next(state, d);
          oplusTo(next[tight_][state_], otimes(curr[tight][state], d));
        }
      }
    }
    curr = next;
  }
  Maybe ans;
  for (int tight = 0; tight <= eq; ++tight)
    for (int state = 0; state < M.size(); ++state)
      if (M.accept(state)) oplusTo(ans, curr[tight][state]);
  return ans.value;
}

template <class T>
string toString(T x) {
  stringstream ss;
  ss << x;
  return ss.str();
}

// Sum of digits from a to b
using Int = long long;
Int sumOfDigits(string z, bool eq = true) {
  struct Value {
    Int count, sum;
    Value &operator+=(Value y) { count+=y.count; sum+=y.sum; return *this; }
    Value &operator*=(int d) { sum+=count*d; return *this; }
  };
  struct Automaton {
    int init = 0;
    int size() { return 1; }
    int next(int s, int d) { return 0; }
    int accept(int s) { return true; }
  };
  return digitDP(z, (Value){1,0}, Automaton(), eq).sum;
}

void SPOJ_CPCRC1C() {
  for (long long a, b; cin >> a >> b; ) {
    if (a < 0 && b < 0) break;
    cout << sumOfDigits(toString(b), true)
        - sumOfDigits(toString(a), false) << endl;
  }
}

//
// Count the zigzag numbers that is a multiple of M.
// Here, a number is zigzag if its digits are alternatively
// increasing and decreasing, like 14283415...

struct Automaton {
  vector<vector<int>> trans;
  vector<bool> is_accept;
  int init = 0;
  int next(int state, int a) { return trans[state][a]; }
  bool accept(int state) { return is_accept[state]; }
  int size() { return trans.size(); }
};

template <class Automaton1, class Automaton2>
Automaton intersectionAutomaton(Automaton1 A, Automaton2 B) {
  Automaton M;
  vector<vector<int>> table(A.size(), vector<int>(B.size(), -1));
  vector<int> x = {A.init}, y = {B.init};
  table[x[0]][y[0]] = 0;
  for (int i = 0; i < x.size(); ++i) {
    M.trans.push_back(vector<int>(10, -1));
    M.is_accept.push_back(A.accept(x[i]) && B.accept(y[i]));
    for (int a = 0; a <= 9; ++a) {
      int u = A.next(x[i], a), v = B.next(y[i], a);
      if (table[u][v] == -1) {
        table[u][v] = x.size();
        x.push_back(u);
        y.push_back(v);
      }
      M.trans[i][a] = table[u][v];
    }
  }
  return M;
}

void AOJ_ZIGZAG() {
  char A[1000], B[1000];
  int M;
  scanf("%s %s %d", A, B, &M);

  struct Value {
    int value = 0;
    Value &operator+=(Value x) {
      if ((value += x.value) >= 10000) value -= 10000;
      return *this;
    }
```

```
    Value &operator*=(int d) {
      return *this;
    }
  } e = (Value){1};

  struct ZigZagAutomaton {
    int init = 0;
    int size() { return 29; }
    int next(int state, int a) {
      if (state == 0) return a == 0 ? 0 : a + 1;
      if (state == 1) return 1;
      if (state <= 10) {
        int last = state - 1;
        if      (a > last) return a + 10;
        else if (a < last) return a + 20;
      } else if (state <= 19) {
        int last = state - 10;
        if (a < last) return a + 20;
      } else if (state <= 28) {
        int last = state - 20;
        if (a > last) return a + 10;
      }
      return 1;
    }
    bool accept(int state) { return state != 1; }
  } zigzag;

  // state =  x : x == n % mod
  struct ModuloAutomaton {
    int mod;
    ModuloAutomaton(int mod) : mod(mod) { }
    int init = 0;
    int size() { return mod; }
    int next(int state, int a) { return (10 * state + a) % mod; }
    bool accept(int state) { return state == 0; }
  } modulo(M);

  auto IM = intersectionAutomaton(zigzag, modulo);
  int a = digitDP(A, e, IM, 0).value;
  int b = digitDP(B, e, IM, 1).value;
  cout << (b + (10000 - a)) % 10000 << endl;
}

//
// Count the numbers that does not contain 4 and 7 in each digit.
// from a to b
void ABC007D() {
  string a, b;
  cin >> a >> b;

  struct ForbiddenNumber {
    int init = 0;
    int size() { return 2; }
    int next(int state, int a) {
      if (state == 1) return 1;
      if (a == 4 || a == 7) return 1;
      return 0;
    }
    bool accept(int state) { return state == 1; }
  };
  struct Counter {
    long long value = 0;
    Counter &operator+=(Counter x) {
      value += x.value;
      return *this;
    }
    Counter &operator*=(int d) {
      return *this;
    }
  };
  cout << digitDP(b, (Counter){1}, ForbiddenNumber(), true).value
      - digitDP(a, (Counter){1}, ForbiddenNumber(), false).value << endl;
}
```

6.11 Edit distance

```
// Minimum number of operations (insert, remove, replace)
// to make strings equal
// O(n^2)

int editDistDP( string s1, string s2 ){
  int m = s1.size(), n = s2.size();
  int dp[m+1][n+1];
  for( int i = 0 ; i <= n ; ++i ){
    for( int j = 0 ; j <= m ; ++j ){
      if( i == 0 ) dp[i][j] = j;
      else if( j == 0 ) dp[i][j] = i;
      else if( s1[i-1] == s2[j-1] )
        dp[i][j] = dp[i-1][j-1];
      else
        //insert, remove, replace respectively
        dp[i][j] = 1 + min( dp[i][j-1], min( dp[i-1][j], dp[i-1][j-1] ) );
    }
  }
  return dp[n][m];
}
```

7 Geometry

7.1 Klee (Area of intersection of rects)

```
// Area of intersecting rectangles
// O(n log n)
#define ll long long

struct rect {
  int x1, y1, x2, y2;
};

class footprint_segtree {
  const int N;
  const vector<int>& weights;
  vector<int> mi, cnt, lazy;
  int total;

  void init(int lo, int hi, int node) {
    if (lo == hi) {
      cnt[node] = weights[lo];
      total += cnt[node];
      return;
    }
    int mid = (lo + hi) / 2;
    init(lo, mid, 2 * node + 1);
    init(mid + 1, hi, 2 * node + 2);
    cnt[node] = cnt[2 * node + 1] + cnt[2 * node + 2];
  }

  void push(int lo, int hi, int node) {
    if (lazy[node]) {
      mi[node] += lazy[node];
      if (lo != hi) {
```

```
        lazy[2 * node + 1] += lazy[node];
        lazy[2 * node + 2] += lazy[node];
      }
      lazy[node] = 0;
    }
  }

  void update_range(int s, int e, int x, int lo, int hi, int node) {
    push(lo, hi, node);
    if (lo > e || hi < s)
      return;
    if (s <= lo && hi <= e) {
      lazy[node] = x;
      push(lo, hi, node);
      return;
    }
    int mid = (lo + hi) / 2;
    update_range(s, e, x, lo, mid, 2 * node + 1);
    update_range(s, e, x, mid + 1, hi, 2 * node + 2);

    mi[node] = min(mi[2 * node + 1], mi[2 * node + 2]);
    cnt[node] = 0;
    if (mi[2 * node + 1] == mi[node])
      cnt[node] += cnt[2 * node + 1];
    if (mi[2 * node + 2] == mi[node])
      cnt[node] += cnt[2 * node + 2];
  }

public:
  footprint_segtree(const vector<int>& weights)
    : N(weights.size()), weights(weights) {
    mi.resize(4 * N);
    cnt.resize(4 * N);
    lazy.resize(4 * N);
    total = 0;
    init(0, N - 1, 0);
  }

  void update_range(int s, int e, int x) {
    update_range(s, e, x, 0, N - 1, 0);
  }

  int query() {
    return total - (mi[0] ? 0 : cnt[0]);
  }
};

ll rectangle_union(const vector<rect>& rects) {
  // Coordinate Compression
  vector<int> ys;
  for (const rect& r : rects) {
    ys.push_back(r.y1);
    ys.push_back(r.y2);
  }
  sort(ys.begin(), ys.end());
  ys.resize(unique(ys.begin(), ys.end()) - ys.begin());

  vector<int> lengths(ys.size() - 1);
  for (int i = 0; i + 1 < ys.size(); i++)
    lengths[i] = ys[i + 1] - ys[i];
  footprint_segtree st(lengths);

  // Sweepline Preparation
  vector<pair<int, pair<int, int> > > events;
  for (int i = 0; i < rects.size(); i++) {
    const rect& r = rects[i];
    events.push_back({ r.x1, { i, 1 } });
    events.push_back({ r.x2, { i, -1 } });
  }
  sort(events.begin(), events.end());

  // Sweepline
  int pre = INT_MIN;
  ll ret = 0;
  for (auto& e : events) {
    ret += (ll) st.query() * (e.first - pre);
    pre = e.first;

    const rect& r = rects[e.second.first];
    int change = e.second.second;
    int y1 = lower_bound(ys.begin(), ys.end(), r.y1) - ys.begin();
    int y2 = lower_bound(ys.begin(), ys.end(), r.y2) - ys.begin();
    st.update_range(y1, y2 - 1, change);
  }

  return ret;
}
```

7.2 Convex hull

```
// O(n log n )
// NAO ESQUECE QUE O TAMANHO DO HULL VAI MUDAR, NAO USE N, USE .size()
// COLOQUEI UM n POR PARAMETRO PRA ISSO, MAS SE VAI USAR O N ANTIGO NAO PASSE
// #CUIDADO
typedef pair<double, double> point;
double ccw( point a, point b, point c ) {
    return (b.first - a.first) * (c.second - a.second ) - (b.second - a.second)
        * (c.first - a.first );
}

vector<point> ch( point *points, int &n ) {
  sort( points, points+n );
  vector<point> hull( n + 1 );
  int idx = 0;
  for( int i = 0 ; i < n ; ++i ) {
    while( idx >= 2 && ccw( hull[idx - 2], hull[idx - 1], points[i] ) >= 0 ) --
        idx;
    hull[idx++] = points[i];
  }
  int half = idx;
  for( int i = n - 2 ; i >= 0 ; --i ) {
    while( idx > half && ccw( hull[idx - 2], hull[idx - 1], points[i] ) >= 0 )
        --idx;
    hull[idx++] = points[i];
  }
  --idx;
  hull.resize( idx );
  n = hull.size();
  return hull;
}
```

7.3 Closest pair with line sweep

```
// Closest pair with line sweep
// O(n log n)
#define ll long long
#define nd second
#define st first
int n; //amount of points
pair<ll, ll> pnt[MAXN];

struct cmp{
  bool operator()(pair<ll,ll> a, pair<ll, ll> b) { return a.nd < b.nd; }
```

```
};

double closest_pair() {
  sort( pnt, pnt + n );
  double best = numeric_limits<double>::infinity();
  set<pair<ll, ll>, cmp> box;
  box.insert( pnt[0] );
  int l = 0;
  for( int i = 1 ; i < n ; ++i ){
    while( l < i && pnt[i].st - pnt[l].st > best )
      box.erase( pnt[l++] );
    for( auto it = box.lower_bound( {0, pnt[i].nd - best} ) ; it != box.end() &&
          pnt[i].nd + best >= it->nd ; ++it )
      best = min( best, hypot( pnt[i].st - it->st, pnt[i].nd - it->nd ) );
    box.insert( pnt[i] );
  }
  return best;
}
```

7.4 Point2D

```
template <class T> int sgn( T x ) { return (x > 0) - (x < 0); }
template<class T> struct Point {
  typedef Point P;
  T x, y;
  explicit Point(T x=0, T y=0) : x(x), y(y) {}
  explicit Point(const Point &a, const Point &b) : x(b.x - a.x), y(b.y - a.y) {}
  bool operator<(P p) const { return tie(x,y) < tie(p.x,p.y); }
  bool operator==(P p) const { return tie(x,y)==tie(p.x,p.y); }
  P operator+(P p) const { return P(x+p.x, y+p.y); }
  P operator-(P p) const { return P(x-p.x, y-p.y); }
  P operator*(T d) const { return P(x*d, y*d); }
  P operator/(T d) const { return P(x/d, y/d); }
  T dot(P p) const { return x*p.x + y*p.y; }
  T cross(P p) const { return x*p.y - y*p.x; }
  T cross(P a, P b) const { return (a-*this).cross(b-*this); }
  T dist2() const { return x*x + y*y; }
  double dist() const { return sqrt((double)dist2()); }
  // angle to x-axis in interval [-pi, pi]
  double angle() const { return atan2(y, x); }
  P unit() const { return *this/dist(); }
  P perp() const { return P(-y, x); }
  P normal() const { return perp().unit(); }
  // returns point rotated 'a' radians ccw around the origin
  P rotate(double a) const { return P(x*cos(a)-y*sin(a),x*sin(a)+y*cos(a)); }
};
```

7.5 Line distance

```
/**
Returns the signed distance between point p and the line containing points a and
     b. Positive value on left side and negative on right as seen from a
     towards b. a==b gives nan. P is supposed to be Point<T> or Point3D<T> where
      T is e.g. double or long long. It uses products in intermediate steps so
     watch out for overflow if using int or long long. Using Point3D will always
      give a non-negative distance.
O(1)
*/
#include "point.cpp"

template<class P>
double lineDist(const P& a, const P& b, const P& p) {
  return (double)(b-a).cross(p-a)/(b-a).dist();
}
```

```
// from point p to seg b-a
double dist( P p, P a, P b ) {
  double k = ((p-a).dot(b-a))/((b-a).dot(b-a));
  return hypot( a.x+(b-a).x*k - p.x, a.y + (b-a).y*k - p.y );
}

// check if three points are collinear (integer)
bool collinear( P p1, P p2, P p3 ) {
  return (p1.y-p2.y) * (p1.x - p3.x) == (p1.y - p3.y) * (p1.x - p2.x );
}
//double
bool collinear(P p1, P p2, P p3 ) {
  return fabs((p1.y - p2.y) * (p1.x - p3.x) - (p1.y - p3.y) * (p1.x - p2.x)) <=
       1e-9;
}
```

7.6 Side of point from segment

```
/**
bool left = sideOf(p1,p2,q)==1;
O(1)
 */
#include "point.cpp"

template<class P>
int sideOf(P s, P e, P p) { return sgn(s.cross(e, p)); }

template<class P>
int sideOf(const P& s, const P& e, const P& p, double eps) {
  auto a = (e-s).cross(p-s);
  double l = (e-s).dist()*eps;
  return (a > l) - (a < -l);
}
```

7.7 Closest distance to segment

```
/**
Returns the shortest distance between point p and the line segment from point s
     to e.
bool onSegment = segDist(a,b,p) < 1e-10;
O(1)
*/
#include "point.cpp"

template<class P> bool onSegment( P a, P b, P c ) {
  return segDist(a,b,c) < 1e-10;
}

typedef Point<double> P;
double segDist(P& s, P& e, P& p) {
  if (s==e) return (p-s).dist();
  auto d = (e-s).dist2(), t = min(d,max(.0,(p-s).dot(e-s)));
  return ((p-s)*d-(e-s)*t).dist()/d;
}
```

7.8 Segment Intersection

```
/**
If a unique intersection point between the line segments going from s1 to e1 and
      from s2 to e2 exists then it is returned.
```

```
If no intersection point exists an empty vector is returned. If infinitely many
     exist a vector with 2 elements is returned, containing the endpoints of the
      common line segment.
The wrong position will be returned if P is Point<ll> and the intersection point
      does not have integer coordinates.
Products of three coordinates are used in intermediate steps so watch out for
     overflow if using int or long long.
vector<P> inter = segInter(s1,e1,s2,e2);
if (sz(inter)==1)
  cout << "segments intersect at " << inter[0] << endl;
O(1)
*/
#pragma once

#include "point.cpp"
#include "segdist.cpp"

template<class P> vector<P> segInter(P a, P b, P c, P d) {
  auto oa = c.cross(d, a), ob = c.cross(d, b),
       oc = a.cross(b, c), od = a.cross(b, d);
  // Checks if intersection is single non-endpoint point.
  if (sgn(oa) * sgn(ob) < 0 && sgn(oc) * sgn(od) < 0)
    return {(a * ob - b * oa) / (ob - oa)};
  set<P> s;
  if (onSegment(c, d, a)) s.insert(a);
  if (onSegment(c, d, b)) s.insert(b);
  if (onSegment(a, b, c)) s.insert(c);
  if (onSegment(a, b, d)) s.insert(d);
  return {all(s)};
}
```

7.9 Line Intersection

```
/**
If a unique intersection point of the lines going through s1,e1 and s2,e2 exists
      \{1, point\} is returned.
If no intersection point exists \{0, (0,0)\} is returned and if infinitely many
     exists \{-1, (0,0)\} is returned.
The wrong position will be returned if P is Point<ll> and the intersection point
      does not have integer coordinates.
Products of three coordinates are used in intermediate steps so watch out for
     overflow if using int or ll.
  auto res = lineInter(s1,e1,s2,e2);
  if (res.first == 1)
    cout << "intersection point at " << res.second << endl;
O(1)
*/
#include "point.cpp"

template<class P>
pair<int, P> lineInter(P s1, P e1, P s2, P e2) {
  auto d = (e1 - s1).cross(e2 - s2);
  if (d == 0) return {-(s1.cross(e1, s2) == 0), P(0, 0)};
  auto p = s2.cross(e1, e2), q = s2.cross(e2, s1);
  return {1, (s1 * p + e1 * q) / d};
}
```

7.10 Tangent points of circle

```
/**
pair of the two points on the circle with radius r centered around c whos
     tangent lines intersect p. If p lies within the circle NaN-points are
     returned. P is intended to be Point<double>. The first point is the one to
     the right as seen from the p towards c.
O(1)
 */
#include "point.cpp"

template<class P>
pair<P,P> circleTangents(const P &p, const P &c, double r) {
  P a = p-c;
  double x = r*r/a.dist2(), y = sqrt(x-x*x);
  return make_pair(c+a*x+a.perp()*y, c+a*x-a.perp()*y);
}
```

7.11 Circumcircle

```
/**
The circumcirle of a triangle is the circle intersecting all three vertices.
ccRadius returns the radius of the circle going through points A, B and C and
ccCenter returns the center of the same circle.
O(1)
 */
#include "point.cpp"

typedef Point<double> P;
double ccRadius(const P& A, const P& B, const P& C) {
  return (B-A).dist()*(C-B).dist()*(A-C).dist()/abs((B-A).cross(C-A))/2;
}
P ccCenter(const P& A, const P& B, const P& C) {
  P b = C-A, c = B-A;
  return A + (b*c.dist2()-c*b.dist2()).perp()/b.cross(c)/2;
}
```

7.12 Circle-Line Intersection

```
// p1 and p2 defines line
// cen is center and rad is radius from circle
// r1, r2 are the points that intersect, returns number of points intersecting
     circle
#include "point.cpp"
#define EPS 1e-9
#ifndef M_PI
#define M_PI 3.14159265358979323846264338327950288 4L
#endif
int circleLineIntersection(const point& p0, const point& p1, const point& cen,
     double rad, point& r1, point & r2) {
  double a, b, c, t1, t2;
  a = (p1 - p0).dot(p1 - p0);
  b = 2 * (p1 - p0).dot(p0 - cen);
  c = (p0-cen).dot(p0-cen) - rad * rad;
  double det = b * b - 4 * a * c;
  int res;
  if( fabs( det ) < EPS ) det = 0, res = 1;
  else if( det < 0 ) res = 0;
  else res = 2;
  det = sqrt( det );
  t1 = ( -b + det ) / ( 2 * a );
  t2 = ( -b - det ) / ( 2 * a );
  r1 = p0 + ( p1 - p0 ) * t1;
  r2 = p0 + ( p1 - p0 ) * t2;
  return res;
}
// returns the arc length
// p1, p2 are the segment
// r radius, cen is center of circle
double calcArc( point p1, point p2, double r, point &cen ) {
  double d = (p2-p1).dist();
```

```
    double ang = ((p1-cen).angle() - (p2-cen).angle()) * 180 / M_PI;
    if( ang < 0 ) ang += 360;
    ang = min( ang, 360 - ang );
    return r * ang * M_PI / 180;
  }
```

7.13 Minimum Enclosing Circle

```
/**
 * Computes the minimum circle that encloses a set of points.
 * O(n) maybe
 */
#include "circumcircle.cpp"

pair<P, double> mec( vector<P> ps ) {
  shuffle(ps.begin(), ps.end(), mt19937(time(0)));
  P o = ps[0];
  double r = 0, EPS = 1 + 1e-8;
  for( int i = 0 ; i < ps.size() ; ++i ) {
    if( (o - ps[i]).dist() > r * EPS ) {
      o = ps[i], r = 0;
      for( int j = 0 ; j < i ; ++j ) {
        if( (o - ps[j]).dist() > r * EPS ) {
          o = (ps[i] + ps[j])/2;
          r = (o - ps[i]).dist();
          for( int k = 0 ; k  < j ; ++k ) {
            if( (o - ps[k]).dist() > r * EPS ) {
              o = ccCenter( ps[i], ps[j], ps[k] );
              r = (o - ps[i]).dist();
            }
          }
        }
      }
    }
  }
  return {o, r};
}
```

7.14 Intersection of two circles

```
/**
pair of points at which two circles intersect.
Returns false in case of no intersection.
O(1)
 */
#include "point.cpp"

typedef Point<double> P;
bool circleInter(P a,P b,double r1,double r2,pair<P, P>* out) {
  if (a == b) { assert(r1 != r2); return false; }
  P vec = b - a;
  double d2 = vec.dist2(), sum = r1+r2, dif = r1-r2,
  p = (d2 + r1*r1 - r2*r2)/(d2*2), h2 = r1*r1 - p*p*d2;
  if (sum*sum < d2 || dif*dif > d2) return false;
  P mid = a + vec*p, per = vec.perp() * sqrt(fmax(0, h2) / d2);
  *out = {mid + per, mid - per};
  return true;
}
```

7.15 Hull Diameter

```
/**
Returns the two points with max distance on a convex hull (ccw, no duplicate/
    colinear points).
O(n) ?
*/
#include<point.cpp>

typedef Point<ll> P;
array<P, 2> hullDiameter(vector<P> S) {
  int n = sz(S), j = n < 2 ? 0 : 1;
  pair<ll, array<P, 2>> res({0, {S[0], S[0]}});
  for( int i = 0 ; i < j ; ++i )
    for (;; j = (j + 1) % n) {
      res = max(res, {(S[i] - S[j]).dist2(), {S[i], S[j]}});
      if ((S[(j + 1) % n] - S[j]).cross(S[i + 1] - S[i]) >= 0)
        break;
    }
  return res.second;
}
```

7.16 Point Inside Polygon

```
/**
 * Returns true if p lies within the polygon. If strict is true,
 * it returns false for points on the boundary. The algorithm uses
 * products in intermediate steps so watch out for overflow.
 * O(n)
 */
#include "point.cpp"
#include "segdist.cpp"

template<class P>
bool inPolygon(vector<P> &p, P a, bool strict = true) {
  int cnt = 0, n = p.size();
  for( int i = 0 ; i < n ; ++i ){
    P q = p[(i + 1) % n];
    if (onSegment(p[i], q, a)) return !strict;
    //or: if (segDist(p[i], q, a) <= eps) return !strict;
    cnt ^= ((a.y<p[i].y) - (a.y<q.y)) * a.cross(p[i], q) > 0;
  }
  return cnt;
}
```

7.17 Point Inside Hull

```
/**
Determine whether a point t lies inside a convex hull (CCW
order, with no colinear points). Returns true if point lies within
the hull. If strict is true, points on the boundary aren't included.
O(\log N)
 */

#include "point.cpp"
#include "sideOf.cpp"
#include "segdist.cpp"

typedef Point<ll> P;

bool inHull(const vector<P>& l, P p, bool strict = true) {
  int a = 1, b = l.size() - 1, r = !strict;
  if (l.size() < 3) return r && onSegment(l[0], l.back(), p);
  if (sideOf(l[0], l[a], l[b]) > 0) swap(a, b);
  if (sideOf(l[0], l[a], p) >= r || sideOf(l[0], l[b], p)<= -r)
    return false;
```

```
  while (abs(a - b) > 1) {
    int c = (a + b) / 2;
    (sideOf(l[0], l[c], p) > 0 ? b : a) = c;
  }
  return sgn(l[a].cross(l[b], p)) < r;
}
```

7.18 Delaunay triangulation

```
//O(n^2)
/*
Computes the Delaunay triangulation of a set of points.
Each circumcircle contains none of the input points.
If any three points are colinear or any four are on the same circle, behavior is
     undefined.
*/
#include "point.cpp"
#include "3dhull.cpp"

template<class P, class F>
void delaunay(vector<P>& ps, F trifun) {
  if (sz(ps) == 3) { int d = (ps[0].cross(ps[1], ps[2]) < 0);
    trifun(0,1+d,2-d); }
  vector<P3> p3;
  trav(p, ps) p3.emplace_back(p.x, p.y, p.dist2());
  if (sz(ps) > 3) trav(t, hull3d(p3)) if ((p3[t.b]-p3[t.a]).
      cross(p3[t.c]-p3[t.a]).dot(P3(0,0,1)) < 0)
    trifun(t.a, t.c, t.b);
}

/**
Each circumcircle contains none of the input points.
There must be no duplicate points.
If all points are on a line, no triangles will be returned.
Should work for doubles as well, though there may be precision issues in 'circ'.
Returns triangles in order \{t[0][0], t[0][1], t[0][2], t[1][0], \dots\}, all
     counter-clockwise.
O(n log n)
*/
#include "point.cpp"

typedef Point<ll> P;
typedef struct Quad* Q;
typedef __int128_t lll; // (can be ll if coords are < 2e4)
P arb(LLONG_MAX,LLONG_MAX); // not equal to any other point

struct Quad {
  bool mark; Q o, rot; P p;
  P F() { return r()->p; }
  Q r() { return rot->rot; }
  Q prev() { return rot->o->rot; }
  Q next() { return r()->prev(); }
};

bool circ(P p, P a, P b, P c) { // is p in the circumcircle?
  lll p2 = p.dist2(), A = a.dist2()-p2,
      B = b.dist2()-p2, C = c.dist2()-p2;
  return p.cross(a,b)*C + p.cross(b,c)*A + p.cross(c,a)*B > 0;
}
Q makeEdge(P orig, P dest) {
  Q q[] = {new Quad{0,0,0,orig}, new Quad{0,0,0,arb},
           new Quad{0,0,0,dest}, new Quad{0,0,0,arb}};
  for( int i = 0 ; i < 4 ; ++i )
    q[i]->o = q[-i & 3], q[i]->rot = q[(i+1) & 3];
  return *q;
}
void splice(Q a, Q b) {
  swap(a->o->rot->o, b->o->rot->o); swap(a->o, b->o);
}
Q connect(Q a, Q b) {
  Q q = makeEdge(a->F(), b->p);
  splice(q, a->next());
  splice(q->r(), b);
  return q;
}

pair<Q,Q> rec(const vector<P>& s) {
  if (s.size() <= 3) {
    Q a = makeEdge(s[0], s[1]), b = makeEdge(s[1], s.back());
    if (s.size() == 2) return { a, a->r() };
    splice(a->r(), b);
    auto side = s[0].cross(s[1], s[2]);
    Q c = side ? connect(b, a) : 0;
    return {side < 0 ? c->r() : a, side < 0 ? c : b->r() };
  }

#define H(e) e->F(), e->p
#define valid(e) (e->F().cross(H(base)) > 0)
  Q A, B, ra, rb;
  int half = s.size() / 2;
  tie(ra, A) = rec({s.begin(), s.end() - half});
  tie(B, rb) = rec({s.size() - half + s.begin(), s.end()});
  while ((B->p.cross(H(A)) < 0 && (A = A->next())) ||
         (A->p.cross(H(B)) > 0 && (B = B->r()->o)));
  Q base = connect(B->r(), A);
  if (A->p == ra->p) ra = base->r();
  if (B->p == rb->p) rb = base;

#define DEL(e, init, dir) Q e = init->dir; if (valid(e)) \
    while (circ(e->dir->F(), H(base), e->F())) { \
      Q t = e->dir; \
      splice(e, e->prev()); \
      splice(e->r(), e->r()->prev()); \
      e = t; \
    }
  for (;;) {
    DEL(LC, base->r(), o);  DEL(RC, base, prev());
    if (!valid(LC) && !valid(RC)) break;
    if (!valid(LC) || (valid(RC) && circ(H(RC), H(LC))))
      base = connect(RC, base->r());
    else
      base = connect(base->r(), LC->r());
  }
  return { ra, rb };
}

vector<P> triangulate(vector<P> pts) {
  sort(pts.begin(), pts.end());
  if (pts.size() < 2) return {};
  Q e = rec(pts).first;
  vector<Q> q = {e};
  int qi = 0;
  while (e->o->F().cross(e->F(), e->p) < 0) e = e->o;
#define ADD { Q c = e; do { c->mark = 1; pts.push_back(c->p); \
  q.push_back(c->r()); c = c->next(); } while (c != e); }
  ADD; pts.clear();
  while (qi < q.size()) if (!(e = q[qi++])->mark) ADD;
  return pts;
}
```

7.19 Polygon cut

```
/**
```

```
Returns a vector with the vertices of a polygon with everything to the left of
    the line going from s to e cut away.
vector<P> p = ...;
p = polygonCut(p, P(0,0), P(1,0));
*/
#include "point.cpp"
#include "lineIntersection.cpp"

typedef Point<double> P;
vector<P> polygonCut(const vector<P>& poly, P s, P e) {
  vector<P> res;
  for( int i = 0 ; i < poly.size() ; ++i ) {
    P cur = poly[i], prev = i ? poly[i-1] : poly.back();
    bool side = s.cross(e, cur) < 0;
    if (side != (s.cross(e, prev) < 0))
      res.push_back(lineInter(s, e, cur, prev).second);
    if (side)
      res.push_back(cur);
  }
  return res;
}
```

7.20 Area of polygon

```
/**
Description: Returns twice the signed area of a polygon.
Clockwise enumeration gives negative area. Watch out for overflow if using int
    as T!
O(n)
*/
#include "point.cpp"

template<class T>
T polygonArea2(vector<Point<T>>& v) {
  T a = v.back().cross(v[0]);
  for( int i = 0 ; i < v.size()-1 ; ++i ) a += v[i].cross(v[i+1]);
  return a;
}
```

7.21 Center of polygon

```
/**
center of mass for a polygon.
O(n)
 */
#include "point.cpp"

typedef Point<double> P;
P polygonCenter(const vector<P>& v) {
  P res(0, 0); double A = 0;
  for (int i = 0, j = v.size() - 1; i < v.size(); j = i++) {
    res = res + (v[i] + v[j]) * v[j].cross(v[i]);
    A += v[j].cross(v[i]);
  }
  return res / A / 3;
}
```

7.22 Line convex polygon intersection

```
/**
Line-convex polygon intersection. The polygon must be ccw and have no colinear
    points.
 * lineHull(line, poly) returns a pair describing the intersection of a line
     with the polygon:
 *    (-1, -1) if no collision,
 *    (i, -1) if touching the corner i,
 *    (i, i) if along side (i, i+1),
 *    (i, j) if crossing sides (i, i+1) and (j, j+1).
In the last case, if a corner $i$ is crossed, this is treated as happening on
    side (i, i+1).
The points are returned in the same order as the line hits the polygon.
extrVertex: returns the point of a hull with the max projection onto a line.
 * Time: O(N + Q \log n)
 */
#include "point.cpp"

typedef array<P, 2> Line;
#define cmp(i,j) sgn(dir.perp().cross(poly[(i)%n]-poly[(j)%n]))
#define extr(i) cmp(i + 1, i) >= 0 && cmp(i, i - 1 + n) < 0
int extrVertex(vector<P>& poly, P dir) {
  int n = sz(poly), lo = 0, hi = n;
  if (extr(0)) return 0;
  while (lo + 1 < hi) {
    int m = (lo + hi) / 2;
    if (extr(m)) return m;
    int ls = cmp(lo + 1, lo), ms = cmp(m + 1, m);
    (ls < ms || (ls == ms && ls == cmp(lo, m)) ? hi : lo) = m;
  }
  return lo;
}

#define cmpL(i) sgn(line[0].cross(poly[i], line[1]))
array<int, 2> lineHull(Line line, vector<P> poly) {
  int endA = extrVertex(poly, (line[0] - line[1]).perp());
  int endB = extrVertex(poly, (line[1] - line[0]).perp());
  if (cmpL(endA) < 0 || cmpL(endB) > 0)
    return {-1, -1};
  array<int, 2> res;
  for( int i = 0 ; i < 2 ; ++i ) {
    int lo = endB, hi = endA, n = sz(poly);
    while ((lo + 1) % n != hi) {
      int m = ((lo + hi + (lo < hi ? 0 : n)) / 2) % n;
      (cmpL(m) == cmpL(endB) ? lo : hi) = m;
    }
    res[i] = (lo + !cmpL(hi)) % n;
    swap(endA, endB);
  }
  if (res[0] == res[1]) return {res[0], -1};
  if (!cmpL(res[0]) && !cmpL(res[1]))
    switch ((res[0] - res[1] + sz(poly) + 1) % sz(poly)) {
      case 0: return {res[0], res[0]};
      case 2: return {res[1], res[1]};
    }
  return res;
}
```

7.23 Volume of polyhedron

```
/**
Faces should point outwards.
O(n)
*/
template<class V, class L>
double signed_poly_volume(const V& p, const L& trilist) {
  double v = 0;
  for( auto i : trilist ) v += p[i.a].cross(p[i.b]).dot(p[i.c]);
  return v / 6;
}
```

7.24 Linear Transformation

```
/**
Apply the linear transformation (translation, rotation and scaling) which takes
     line p0-p1 to line q0-q1 to point r.
O(1)
*/
#include "point.cpp"

typedef Point<double> P;
P transform(const P& p0, const P& p1, const P& q0, const P& q1, const P& r) {
  P dp = p1-p0, dq = q1-q0, num(dp.cross(dq), dp.dot(dq));
  return q0 + P((r-p0).cross(num), (r-p0).dot(num))/dp.dist2();
}
```

7.25 Spherical Distance

```
/**
Returns the shortest distance on the sphere with radius radius between the
     points
with azimuthal angles (longitude) f1 and f2 from x axis and zenith angles (
     latitude)
t1 and t2 from z axis. All angles measured in radians. The algorithm starts by
converting the spherical coordinates to cartesian coordinates so if that is what
you have you can use only the two last rows. dx*radius is then the difference
     between
the two points in the x direction and d*radius is the total distance between the
      points.
*/
double sphericalDistance(double f1, double t1, double f2, double t2, double
     radius) {
  double dx = sin(t2)*cos(f2) - sin(t1)*cos(f1);
  double dy = sin(t2)*sin(f2) - sin(t1)*sin(f1);
  double dz = cos(t2) - cos(t1);
  double d = sqrt(dx*dx + dy*dy + dz*dz);
  return radius*2*asin(d/2);
}
```

7.26 Angle sorting

```
/**
Description: A class for ordering angles (as represented by int points and
a number of rotations around the origin). Useful for rotational sweeping.
Sometimes also represents points or vectors.
Usage:
vector<Angle> v = {w[0], w[0].t360() ...}; // sorted
int j = 0; rep(i,0,n) { while (v[j] < v[i].t180()) ++j; }
// sweeps j such that (j-i) represents the number of positively oriented
     triangles with vertices at 0 and i
*/
struct Angle {
  int x, y;
  int t;
  Angle(int x, int y, int t=0) : x(x), y(y), t(t) {}
  Angle operator-(Angle b) const { return {x-b.x, y-b.y, t}; }
  int half() const {
    assert(x || y);
    return y < 0 || (y == 0 && x < 0);
  }
  Angle t90() const { return {-y, x, t + (half() && x >= 0)}; }
  Angle t180() const { return {-x, -y, t + half()}; }
  Angle t360() const { return {x, y, t + 1}; }
};
bool operator<(Angle a, Angle b) {
  // add a.dist2() and b.dist2() to also compare distances
  return make_tuple(a.t, a.half(), a.y * (ll)b.x) <
         make_tuple(b.t, b.half(), a.x * (ll)b.y);
}

// Given two points, this calculates the smallest angle between
// them, i.e., the angle that covers the defined line segment.
pair<Angle, Angle> segmentAngles(Angle a, Angle b) {
  if (b < a) swap(a, b);
  return (b < a.t180() ?
          make_pair(a, b) : make_pair(b, a.t360()));
}
Angle operator+(Angle a, Angle b) { // point a + vector b
  Angle r(a.x + b.x, a.y + b.y, a.t);
  if (a.t180() < r) r.t--;
  return r.t180() < a ? r.t360() : r;
}
Angle angleDiff(Angle a, Angle b) { // angle b - angle a
  int tu = b.t - a.t; a.t = b.t;
  return {a.x*b.x + a.y*b.y, a.x*b.y - a.y*b.x, tu - (b < a)};
}
```

7.27 K-D Tree

```
/**
find the nearest neighbour of a point O(logn) on average
*/
#include "point.cpp"

typedef long long T;
typedef Point<T> P;
const T INF = numeric_limits<T>::max();

bool on_x(const P& a, const P& b) { return a.x < b.x; }
bool on_y(const P& a, const P& b) { return a.y < b.y; }

struct Node {
  P pt; // if this is a leaf, the single point in it
  T x0 = INF, x1 = -INF, y0 = INF, y1 = -INF; // bounds
  Node *first = 0, *second = 0;

  T distance(const P& p) { // min squared distance to a point
    T x = (p.x < x0 ? x0 : p.x > x1 ? x1 : p.x);
    T y = (p.y < y0 ? y0 : p.y > y1 ? y1 : p.y);
    return (P(x,y) - p).dist2();
  }

  Node(vector<P>&& vp) : pt(vp[0]) {
    for (P p : vp) {
      x0 = min(x0, p.x); x1 = max(x1, p.x);
      y0 = min(y0, p.y); y1 = max(y1, p.y);
    }
    if (vp.size() > 1) {
      // split on x if the box is wider than high (not best heuristic...)
      sort(vp.begin(), vp.end(), x1 - x0 >= y1 - y0 ? on_x : on_y);
      // divide by taking half the array for each child (not
      // best performance with many duplicates in the middle)
      int half = sz(vp)/2;
      first = new Node({vp.begin(), vp.begin() + half});
      second = new Node({vp.begin() + half, vp.end()});
    }
  }
};

struct KDTree {
```

```
  Node* root;
  KDTree(const vector<P>& vp) : root(new Node({vp.begin(), vp.end()})) {}

  pair<T, P> search(Node *node, const P& p) {
    if (!node->first) {
      // uncomment if we should not find the point itself:
      // if (p == node->pt) return {INF, P()};
      return make_pair((p - node->pt).dist2(), node->pt);
    }

    Node *f = node->first, *s = node->second;
    T bfirst = f->distance(p), bsec = s->distance(p);
    if (bfirst > bsec) swap(bsec, bfirst), swap(f, s);

    // search closest side first, other side if needed
    auto best = search(f, p);
    if (bsec < best.first)
      best = min(best, search(s, p));
    return best;
  }

  // find nearest point to a point, and its squared distance
  // (requires an arbitrary operator< for Point)
  pair<T, P> nearest(const P& p) {
    return search(root, p);
  }
};
```

7.28 Point3D

```
template<class T> struct Point3D {
  typedef Point3D P;
  typedef const P& R;
  T x, y, z;
  explicit Point3D(T x=0, T y=0, T z=0) : x(x), y(y), z(z) {}
  bool operator<(R p) const { return tie(x, y, z) < tie(p.x, p.y, p.z); }
  bool operator==(R p) const { return tie(x, y, z) == tie(p.x, p.y, p.z); }
  P operator+(R p) const { return P(x+p.x, y+p.y, z+p.z); }
  P operator-(R p) const { return P(x-p.x, y-p.y, z-p.z); }
  P operator*(T d) const { return P(x*d, y*d, z*d); }
  P operator/(T d) const { return P(x/d, y/d, z/d); }
  T dot(R p) const { return x*p.x + y*p.y + z*p.z; }
  P cross(R p) const { return P(y*p.z - z*p.y, z*p.x - x*p.z, x*p.y - y*p.x); }
  T dist2() const { return x*x + y*y + z*z; }
  double dist() const { return sqrt((double)dist2()); }
  //Azimuthal angle (longitude) to x-axis in interval [-pi, pi]
  double phi() const { return atan2(y, x); }
  //Zenith angle (latitude) to the z-axis in interval [0, pi]
  double theta() const { return atan2(sqrt(x*x+y*y),z); }
  P unit() const { return *this/(T)dist(); } //makes dist()=1
  //returns unit vector normal to *this and p
  P normal(P p) const { return cross(p).unit(); }
  //returns point rotated 'angle' radians ccw around axis
  P rotate(double angle, P axis) const {
    double s = sin(angle), c = cos(angle); P u = axis.unit();
    return u*dot(u)*(1-c) + (*this)*c - cross(u)*s;
  }
};
```

7.29 Convex hull 3D

```
// O(n^3) ?
typedef Point3D<double> P3;

struct PR {
  void ins(int x) { (a == -1 ? a : b) = x; }
  void rem(int x) { (a == x ? a : b) = -1; }
  int cnt() { return (a != -1) + (b != -1); }
  int a, b;
};

struct F { P3 q; int a, b, c; };

vector<F> hull3d(const vector<P3>& A) {
  vector<vector<PR>> E(A.size(), vector<PR>(A.size(), {-1, -1}));
#define E(x,y) E[f.x][f.y]
  vector<F> FS;
  auto mf = [&](int i, int j, int k, int l) {
    P3 q = (A[j] - A[i]).cross((A[k] - A[i]));
    if (q.dot(A[l]) > q.dot(A[i]))
      q = q * -1;
    F f{q, i, j, k};
    E(a,b).ins(k); E(a,c).ins(j); E(b,c).ins(i);
    FS.push_back(f);
  };
  for( int i = 0 ; i < 4 ; ++i )
    for( int j = i + 1 ; j < 4 ; ++j )
      for( int k = k + 1 ; k < 4 ; ++k )
        mf(i, j, k, 6 - i - j - k);

  for( int i = 4 ; i < A.size() ; ++i ) {
    for( int j = 0 ; j < FS.size() ; ++j ) {
      F f = FS[j];
      if(f.q.dot(A[i]) > f.q.dot(A[f.a])) {
        E(a,b).rem(f.c);
        E(a,c).rem(f.b);
        E(b,c).rem(f.a);
        swap(FS[j--], FS.back());
        FS.pop_back();
      }
    }
    for( int j = 0 ; j < FS.size() ; ++j ) {
      F f = FS[j];
#define C(a, b, c) if (E(a,b).cnt() != 2) mf(f.a, f.b, i, f.c);
      C(a, b, c); C(a, c, b); C(b, c, a);
    }
  }
  for( auto it : FS )
    if( (A[it.b] - A[it.a]).cross( A[it.c] - A[it.a] ).dot(it.q) <= 0 )
      swap(it.c, it.b);
  return FS;
};
```

7.30 Another geometry lib

```
const double EPS = 1e-9;


struct Point {
  double x, y;

  Point() {}
  Point(double x, double y) : x(x), y(y) {}
  Point(const Point &a, const Point &b) : x(b.x - a.x), y(b.y - a.y) {}

  double angle() const {
    double a = atan2(y, x);
    if (a < -EPS)
      a += 2 * acos(-1.0);
    return a;
  }
```

```cpp
  double length() const {
    return sqrt(x * x + y * y);
  }
  double distanceTo(const Point &that) const {
    return Point(*this, that).length();
  }

  Point operator + (const Point &that) const {
    return Point(x + that.x, y + that.y);
  }
  Point operator - (const Point &that) const {
    return Point(x - that.x, y - that.y);
  }
  Point operator * (double k) const {
    return Point(x * k, y * k);
  }
  Point setLength(double newLength) const {
    double k = newLength / length();
    return Point(x * k, y * k);
  }

  double dotProduct(const Point &that) const {
    return x * that.x + y * that.y;
  }
  double angleTo(const Point &that) const {
    return acos(max(-1.0, min(1.0, dotProduct(that) / (length() * that.length())
        )));
  }
  bool isOrthogonalTo(const Point &that) const {
    return fabs(dotProduct(that)) < EPS;
  }
  Point orthogonalPoint() const {
    return Point(-y, x);
  }

  double crossProduct(const Point &that) const {
    return x * that.y - y * that.x;
  }
  bool isCollinearTo(const Point &that) const {
    return fabs(crossProduct(that)) < EPS;
  }
};


struct Line {
  double a, b, c;

  Line() {}
  Line(double a, double b, double c) : a(a), b(b), c(c) {}
  Line(const Point &p1, const Point &p2) : a(p1.y - p2.y), b(p2.x - p1.x), c(p1.
      x * p2.y - p2.x * p1.y) {}
  static Line LineByVector(const Point &p, const Point &v) {
    return Line(p, p + v);
  }
  static Line LineByNormal(const Point &p, const Point &n) {
    return LineByVector(p, n.orthogonalPoint());
  }

  Point normal() const {
    return Point(a, b);
  }
  Line orthogonalLine(const Point &p) const {
    return LineByVector(p, normal());
  }
  Line parallelLine(const Point &p) const {
    return LineByNormal(p, normal());
  }
  Line parallelLine(double distance) const {
    Point p = (fabs(a) < EPS ? Point(0, -c / b) : Point(-c / a, 0));
    Point p1 = p + normal().setLength(distance);
    return LineByNormal(p1, normal());
  }

  int side(const Point &p) const {
    double r = a * p.x + b * p.y + c;
    if (fabs(r) < EPS)
      return 0;
    else
      return r > 0 ? 1 : -1;
  }
  double distanceTo(const Point &p) const {
    return fabs(a * p.x + b * p.y + c) / sqrt(a * a + b * b);
  }
  bool has(const Point &p) const {
    return distanceTo(p) < EPS;
  }

  double distanceTo(const Line &that) const {
    if (normal().isCollinearTo(that.normal())) {
      Point p = (fabs(a) < EPS ? Point(0, -c / b) : Point(-c / a, 0));
      return that.distanceTo(p);
    } else
      return 0;
  }
  bool intersectsWith(const Line &that) const {
    return distanceTo(that) < EPS;
  }
  Point intersection(const Line &that) const {
    double d = a * that.b - b * that.a;
    double dx = -c * that.b - b * -that.c;
    double dy = a * -that.c - -c * that.a;
    return Point(dx / d, dy / d);
  }
};


struct Ray {
  Point p1, p2;
  double a, b, c;

  Ray(const Point &p1, const Point &p2) : p1(p1), p2(p2), a(p1.y - p2.y), b(p2.x
      - p1.x), c(p1.x * p2.y - p2.x * p1.y) {}

  double distanceTo(const Point &p) const {
    if (Point(p1, p).dotProduct(Point(p1, p2)) >= -EPS)
      return fabs(a * p.x + b * p.y + c) / sqrt(a * a + b * b);
    else
      return p1.distanceTo(p);
  }
  bool has(const Point &p) const {
    return distanceTo(p) < EPS;
  }

  double distanceTo(const Ray &that) const {
    Line l(a, b, c), thatL(that.a, that.b, that.c);
    if (l.intersectsWith(thatL)) {
      Point p = l.intersection(thatL);
      if (has(p) && that.has(p))
        return 0;
    }
    return min(distanceTo(that.p1), that.distanceTo(p1));
  }
  bool intersectsWith(const Ray &that) const {
    return distanceTo(that) < EPS;
  }
};
```

```
struct Segment {
  Point p1, p2;
  double a, b, c;

  Segment(const Point &p1, const Point &p2) : p1(p1), p2(p2), a(p1.y - p2.y), b(
      p2.x - p1.x), c(p1.x * p2.y - p2.x * p1.y) {}

  double distanceTo(const Point &p) const {
    if (Point(p1, p).dotProduct(Point(p1, p2)) >= -EPS && Point(p2, p).
        dotProduct(Point(p2, p1)) >= -EPS)
      return fabs(a * p.x + b * p.y + c) / sqrt(a * a + b * b);
    else
      return min(p1.distanceTo(p), p2.distanceTo(p));
  }
  bool has(const Point &p) const {
    return distanceTo(p) < EPS;
  }

  double distanceTo(const Segment &that) const {
    Line l(a, b, c), thatL(that.a, that.b, that.c);
    if (l.intersectsWith(thatL)) {
      Point p = l.intersection(thatL);
      if (has(p) && that.has(p))
        return 0;
    }
    return min(min(distanceTo(that.p1), distanceTo(that.p2)), min(that.
        distanceTo(p1), that.distanceTo(p2)));
  }
  bool intersectsWith(const Segment &that) const {
    return distanceTo(that) < EPS;
  }
};


struct Polygon {
  vector<Point> points;
  void addPoint(const Point &p) {
    points.push_back(p);
  }
  double area() const {
    double s = 0;
    for (int i = 1; i < points.size(); i++)
      s += points[i - 1].crossProduct(points[i]);
    s += points[points.size() - 1].crossProduct(points[0]);
    return fabs(s) / 2;
  }
};
```

8 Java

8.1 Template

```
import java.io.IOException;
public class Main {
  public static void main(String[] args) {
    InputStream inputStream = System.in;
    OutputStream outputStream = System.out;
    InputReader in = new InputReader(inputStream);
    PrintWriter out = new PrintWriter(outputStream);
    Task solver = new Task();
    solver.solve(1, in, out);
    out.close();
  }
  static class Task {
    public void solve(int testNumber, InputReader in, PrintWriter out) {

    }
  }
  static class InputReader {
    public BufferedReader reader;
    public StringTokenizer tokenizer;

    public InputReader(InputStream stream) {
      reader = new BufferedReader(new InputStreamReader(stream), 32768);
      tokenizer = null;
    }

    public String next() {
      while (tokenizer == null || !tokenizer.hasMoreTokens()) {
        try {
          tokenizer = new StringTokenizer(reader.readLine());
        } catch (IOException e) {
          throw new RuntimeException(e);
        }
      }
      return tokenizer.nextToken();
    }

    public int nextInt() {
      return Integer.parseInt(next());
    }

  }
}
```

8.2 Big Numbers

```
import java.math.*;
class BMath {
  static int cnt1, cnt2;
  public static MathContext mc = null;
  public static BigDecimal eps = null;
  public static BigDecimal two = null;
  public static BigDecimal sqrt3 = null;
  public static BigDecimal pi = null;
  public static final int PRECISION = 128;
  static {
    mc = new MathContext(PRECISION);
    eps = BigDecimal.ONE.scaleByPowerOfTen(-PRECISION);
    two = BigDecimal.valueOf(2);
    sqrt3 = sqrt(BigDecimal.valueOf(3));
    pi = asin(BigDecimal.valueOf(0.5)).multiply(BigDecimal.valueOf(6));
  }
  public static BigInteger sqrt(BigInteger val) {
    int len = val.bitLength();
    BigInteger left = BigInteger.ONE.shiftLeft((len - 1) / 2);
    BigInteger right = BigInteger.ONE.shiftLeft(len / 2 + 1);
    while (left.compareTo(right) < 0) {
      BigInteger mid = left.add(right).shiftRight(1);
      if (mid.multiply(mid).compareTo(val) <= 0) {
        left = mid.add(BigInteger.ONE);
      } else {
        right = mid;
      }
    }
    return right.subtract(BigInteger.ONE);
  }
  public static BigDecimal sqrt(BigDecimal val) {
    BigInteger unscaledVal = val.scaleByPowerOfTen(2 * mc.getPrecision()).
        toBigInteger();
    return new BigDecimal(sqrt(unscaledVal)).scaleByPowerOfTen(-mc.getPrecision
        ());
```

```
  }
  public static BigDecimal asin(BigDecimal val) {
    BigDecimal tmp = val;
    BigDecimal ret = tmp;
    val = val.multiply(val, mc);
    for (int n = 1; tmp.compareTo(eps) > 0; ++n) {
      tmp = tmp.multiply(val, mc).multiply(
          BigDecimal.valueOf(2 * n - 1).divide(BigDecimal.valueOf(2 * n), mc),
              mc);
      ret = ret.add(tmp.divide(BigDecimal.valueOf(2 * n + 1), mc), mc);
    }
    return ret;
  }
}
```

9 Miscellaneous

9.1 Matrix operations

```
// Matrix arithmetic
#define ll long long
typedef vector<ll> vec;
typedef vector<vec> mat;

const ll MOD = 1e9 + 7;
//O(n^2)
mat zeros( int n, int m )
{
  return mat( n, vec( m ) );
}
//O(n^2)
mat id( int n )
{
  mat ret = zeros( n, n );
  for( int i = 0 ; i < n ; ++i ) ret[i][i] = 1;
  return ret;
}
//O(n^2)
mat add( mat a, const mat& b )
{
  int n = a.size(), m = a[0].size();
  for( int i = 0 ; i < n ; ++i )
    for( int j = 0 ; j < m ; ++j )
      a[i][j] = (a[i][j] + b[i][j]) % MOD;
  return a;
}
//O(n^3)
mat mul( const mat& a, const mat& b )
{
  int n = a.size(), m = a[0].size(), k = b[0].size();
  mat ret = zeros( n, k );
  for( int i = 0 ; i < n ; ++i )
    for( int j = 0 ; j < k ; ++j )
      for( int p = 0 ; p < m ; ++p )
        ret[i][j] = (ret[i][j] + a[i][p] * b[p][j]) % MOD;
  return ret;
}
//O(log n)
mat pow( const mat& a, ll p )
{
  if( p == 0 ) return id( a.size() );
  mat ret = pow( mul( a, a ), p >> 1 );
  if( p & 1 ) ret = mul( ret, a );
  return ret;
}
```

9.2 Good RNG

```
mt19937 get_rng() {
  seed_seq seq {
    (uint64_t) chrono::duration_cast<chrono::nanoseconds>(
      chrono::high_resolution_clock::now().time_since_epoch()).count(),
    (uint64_t) __builtin_ia32_rdtsc(),
    (uint64_t)(uintptr_t) unique_ptr<char>(new char).get()
  };
  return mt19937( seq );
}

int main() {
  auto rng = get_rng();
  uniform_int_distribution<int> distr( 0, 99 );
  cout << distr(rng) << endl;
  return 0;
}
```

9.3 Merge sort with inversions

```
// O(n log n)
#define INF 0x3f3f3f3f
int merge_sort( vector<int> &v ) {
  if( v.size() == 1 ) return 0;
  int inv = 0;
  vector<int> u1, u2;
  for(int i = 0 ; i < v.size() / 2 ; ++i ) u1.push_back(v[i]);
  for( int i = v.size() / 2 ; i < v.size() ; ++i ) u2.push_back( v[i] );
  inv += merge_sort( u1 ) + merge_sort( u2 );
  u1.push_back( INF ), u2.push_back( INF );
  int ini1 = 0, ini2 = 0;
  for( int i = 0 ; i < v.size() ; ++i ){
    if( u1[ini1] <= u2[ini2] )
      v[i]=u1[ini1++];
    else
    {
      v[i] = u2[ini2++];
      inv += u1.size() - ini1 - 1;
    }
  }
  return inv;
}
```

9.4 Fast string to int

```
// O(n)
int fstoi( const char * str ) {
  int val = 0;
  while( *str ) val = val * 10 + ( *str++ - '0' );
  return val;
}
```

9.5 All subsets of a set

```
int b = 0;
do {
  // process subset b
} while( b = ( b - x ) & x );
```

9.6 Convert Parenthesis to Polish

```
inline bool isOp( char c ) {
  return c=='+' || c=='-' || c=='*' || c=='/' || c=='^';
}

inline bool isCarac( char c ) {
  return (c>='a' && c<='z') || (c>='A' && c<='Z') || (c>='0' && c<='9');
}

int paren2polish( char* paren, char* polish ) {
  map<char, int> prec;
  prec['('] = 0;
  prec['+'] = prec['-'] = 1;
  prec['*'] = prec['/'] = 2;
  prec['^'] = 3;
  int len = 0;
  stack<char> op;
  for( int i = 0; paren[i]; ++i ) {
    if( isOp( paren[i] ) ) {
      while( !op.empty() && prec[op.top()] >= prec[paren[i]]) {
        polish[len++] = op.top(); op.pop();
      }
      op.push( paren[i] );
    }
    else if( paren[i]=='(' ) op.push( '(' );
    else if( paren[i]==')' ) {
      for( ; op.top()!='(' ; op.pop() )
        polish[len++] = op.top();
      op.pop();
    }
    else if( isCarac( paren[i] ) )
      polish[len++] = paren[i];
  }
  for( ; !op.empty(); op.pop() ) polish[len++] = op.top();
  polish[len] = 0;
  return len;
}
```

9.7 Week day

```
int v[] = { 0, 3, 2, 5, 0, 3, 5, 1, 4, 6, 2, 4 };
int day( int d, int m, int y ) {
  y -= m < 3;
  return ( y + y / 4 - y / 100 + y / 400 + v[m - 1] + d ) % 7;
}
```

9.8 Latitude-Longitude to rectangular

```
//LatLong <-> rectangular
struct latlong {
  double r, lat, lon;
};
struct rect {
  double x, y, z;
};
latlong convert( rect &P ) {
  latlong Q;
  Q.r = sqrt( P.x * P.x + P.y * P.y + P.z * P.z );
  Q.lat = 180 / M_PI * asin( P.z / Q.r );
  Q.lon = 180 / M_PI * acos( P.x/sqrt( P.x * P.x + P.y * P.y ) );
  return Q;
}

rect convert( latlong &Q )
{
  rect P;
  P.x = Q.r * cos( Q.lon * M_PI / 180 ) * cos( Q.lat * M_PI / 180 );
  P.y = Q.r * sin( Q.lon * M_PI / 180 ) * cos( Q.lat * M_PI / 180 );
  P.z = Q.r * sin( Q.lat * M_PI / 180 );
  return P;
}
```

9.9 Date manipulation

```
struct Date {
  int d, m, y;
  static int mnt[], mntsum[];
  Date() : d( 1 ), m( 1 ), y( 1 ) {}
  Date(int d, int m, int y) : d(d), m(m), y(y) {}
  Date(int days) : d(1), m(1), y(1) { advance(days); }

  bool bissexto() { return (y%4 == 0 and y%100) or (y%400 == 0); }

  int mdays() { return mnt[m] + (m == 2)*bissexto(); }
  int ydays() { return 365+bissexto(); }

  int msum()  { return mntsum[m-1] + (m > 2)*bissexto(); }
  int ysum()  { return 365*(y-1) + (y-1)/4 - (y-1)/100 + (y-1)/400; }

  int count() { return (d-1) + msum() + ysum(); }

  int day() {
    int x = y - (m<3);
    return (x + x/4 - x/100 + x/400 + mntsum[m-1] + d + 6)%7;
  }

  void advance(int days) {
    days += count();
    d = m = 1, y = 1 + days/366;
    days -= count();
    while(days >= ydays()) days -= ydays(), y++;
    while(days >= mdays()) days -= mdays(), m++;
    d += days;
  }
};

int Date::mnt[13] = {0, 31, 28, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31};
int Date::mntsum[13] = {};
for(int i=1; i<13; ++i) Date::mntsum[i] = Date::mntsum[i-1] + Date::mnt[i];
```

9.10 BitHacks

```
// http://www.graphics.stanford.edu/~seander/bithacks.html

template <class T, class X> inline bool getbit(T a, X i) { T t = 1; return ((a &
    (t << i)) > 0);}
template <class T, class X> inline T setbit(T a, X i) { T t = 1; return (a | (t
    << i)); }
template <class T, class X> inline T resetbit(T a, X i) { T t = 1; return (a &
    (~(t << i)));}

__builtin_ctz(x) // trailing zeroes
__builtin_clz(x) // leading zeroes
__builtin_popcount(x) // # bits set
__builtin_ffs(x) // index(LSB) + 1 [0 if x==0]
```

```cpp
bool powerOfTwo( int n ) {
  return n && !( n & ( n - 1 ) );
}

bool opositeSigns( int x, int y ) {
  return ( ( x ^ y ) < 0 );
}

// f true = set, false = clear | m is the bits to change
int changeBit( int n, bool f, int m ) {
  return n = ( n & ~m ) | ( -f & m );
}

//32 bits only (log n)
int reverseBits( int n ) {
  unsigned int s = sizeof( n ) * CHAR_BIT;
  unsigned int mask = ~0;
  while ( ( s >>= 1 ) > 0 )
  {
    mask ^= ( mask << s );
    v = ( ( v >> s ) & mask ) | ( ( v << s ) & ~mask );
  }
  return n;
}

// Round to next power of two (32 bits)
int roundUpP2( int v ) {
  if( v > 1 ) {
    float f = (float)v;
    int const t = 1U << ( ( *( int * ) & f >> 23 ) - 0x7f );
    return t << ( t < v );
  }
  else return 1;
}

// interleave bits, x is even, y is odd (x,y less than 65536)
int interleave( int x, char y ) {
  int z = 0;
  for( int i = 0; i < sizeof(x) * CHAR_BIT; ++i )
    z |= ( x & 1U << i ) << i | ( y & 1U << i ) << ( i + 1 );
  return z;
}

// v is the current permutation (lexicographically)
int next_permutation_bit( int v ) {
  int t = v | (v - 1);
  return( t + 1 ) | ( ( ( ~t & -~t ) - 1 ) >> ( __builtin_ctz( v ) + 1 ) );
}

// check if a word has a byte equal to n
#define hasvalue(x,n) (haszero((x) ^ (~0UL/255 * (n))))
// check if a word has a byte less than n (hasless(n,1) to check if it has a
    zero byte)
#define hasless(x,n) (((x)-~0UL/255*(n))&~(x)&~0UL/255*128)
// check if a word has a byte greater than n
#define hasmore(x,n) (((x)+~0UL/255*(127-(n))|(x))&~0UL/255*128)
```

9.11 Template

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

#define mset( n, v ) memset( n, v, sizeof( n ) )
#define st first
#define nd second
#define INF 0x3f3f3f3f
#define INFLL 0x3f3f3f3f3f3f3f3f
#define pb push_back
#define eb emplace_back
#define PI 3.1415926535897932384626433832795028841L
#define EPS 1e-9
#define mp make_pair
#define sz(x) int(x.size())
#define all(x) x.begin(), x.end()

typedef pair<int, int> pii;
typedef pair<int, ll> pil;
typedef pair<ll, ll> pll;
typedef pair<ll, int> pli;
typedef vector<int> vi;
typedef vector<pii> vpi;
typedef long long ll;
typedef unsigned long long ull;
typedef long double ld;

int main() {
  //fast cin/cout
  ios_base::sync_with_stdio( false );
  cin.tie( 0 );
  freopen("file.in", "r", stdin);
  ofstream fout ("area.out");
  ifstream fin ("area.in");

  // Ouput a specific number of digits past the decimal point,
    // in this case 5
    cout.setf( ios::fixed ); cout << setprecision( 5 );
    cout << 100.0/7.0 << endl;
    cout.unsetf(ios::fixed);

    // Output the decimal point and trailing zeros
    cout.setf( ios::showpoint );
    cout << 100.0 << endl;
    cout.unsetf( ios::showpoint );

    // Output a '+' before positive values
    cout.setf( ios::showpos );
    cout << 100 << " " << -100 << endl;
    cout.unsetf( ios::showpos );

    // Output numerical values in hexadecimal
    cout << hex << 100 << " " << 1000 << " " << 10000 << dec << endl;
  return 0;
}
```

9.12 Difference Array

```cpp
//O(1) range update
//O(n) query

vector<int> initializeDiffArray( vector<int>& A ){
  int n = A.size();
  vector<int> D(n + 1);

  D[0] = A[0], D[n] = 0;
  for (int i = 1; i < n; i++)
    D[i] = A[i] - A[i - 1];
  return D;
}

void update( vector<int>& D, int l, int r, int x ){
  D[l] += x;
  D[r + 1] -= x;
}
```

```cpp
int printArray( vector<int>& A, vector<int>& D ){
  for (int i = 0; i < A.size(); i++) {
    if (i == 0) A[i] = D[i];
    else A[i] = D[i] + A[i - 1];
    cout << A[i] << " ";
  }
  cout << endl;
}
```

9.13 Ternary search

```cpp
double f( double x ) {
  return x;
}

double tsearch( double x ) {
  double l = 0, r = x;
  while( abs( l - r ) > EPS ) {
    double lt = l + ( r - l ) /3;
    double rt = r - ( r - l ) /3;
    if( f(lt) > f(rt) ) l = lt;
    else r = rt;
  }
  return max( r, l );
}
```

9.14 Green Hackenbush

```cpp
// Green hackenbush is a game that each player can cut an edge
// until the root and the player that cant cut anymore loses
// O(n+m)
int n;
vector<int> adj[MAXN];
void add_edge(int u, int v) {
  adj[u].push_back(v);
  if (u != v) adj[v].push_back(u);
}

int grundy(int r) {
  vector<int> num(n), low(n);
  int t = 0;
  function<int(int,int)> dfs = [&](int p, int u) {
    num[u] = low[u] = ++t;
    int ans = 0;
    for (int v: adj[u]) {
      if (v == p) { p += 2*n; continue; }
      if (num[v] == 0) {
        int res = dfs(u, v);
        low[u] = min(low[u], low[v]);
        if (low[v] > num[u]) ans ^= (1 + res) ^ 1;
        else ans ^= res;
      } else low[u] = min(low[u], num[v]);
    }
    if (p > n) p -= 2*n;
    for (int v: adj[u])
      if (v != p && num[u] <= num[v]) ans ^= 1;
    return ans;
  };
  return dfs(-1, r);
}
```

9.15 128 bit integer

```cpp
__int128 input(){
    string s;
    cin >> s;
    ll fst = (s[0] == '-') ? 1 : 0;
    __int128 v = 0;
    f(i,fst,s.size()) v = v * 10 + s[i] - '0';
    if(fst) v = -v;
    return v;
}

ostream& operator << (ostream& os,const __int128& v) {
    string ret, sgn;
    __int128 n = v;
    if(v < 0) sgn = "-", n = -v;
    while(n) ret.pb(n % 10 + '0'), n /= 10;
    reverse(all(ret));
    ret = sgn + ret;
    os << ret;
    return os;
}

int main(){
    __int128 n = input();
    cout << n << endl;
}
```

9.16 Grid Tools

```cpp
#define MAXN 100
int g[MAXN][MAXN],vis[MAXN][MAXN];

/*
CHESS
0 - Horse
1 - Bishop
2 - Rook
3 - Queen
*/

int mod[] = {4,4,3};
vector<vector<int>> chessx = {
    {2, 2, 1, 1, -1, -1, -2, -2},
    {1, 1, -1, -1},
    {1, 0, -1, 0},
    {1, 0, -1, 0, 1, 1, -1, -1}
};

vector<vector<int>> chessy = {
    {1, -1, 2, -2, 2, -2, 1, -1},
    {1, -1, 1, -1},
    {0, 1, 0, -1},
    {0, 1, 0, -1, 1, -1, 1, -1}
};

/*
ROBOT
0 - Clockwise Spiral
1 - CounterClockWise Spiral
2 - Main Diagonal
*/

vector<vector<int>> dx = {
```

```
    {1,0,-1,0},
    {0,1,0,-1},
    {1,0,-1},
};

vector<vector<int>> dy = {
    {0,1,0,-1},
    {1,0,-1,0},
    {1,-1,0},
};



void robot_walk(int i,int j,int t){

    int dir = 0;

    while(!vis[i][j]){

        vis[i][j] = 1;

        if((vis[i+dy[t][dir]][j+dx[t][dir]] ||
          (i+dy[t][dir] >= MAXN || i+dy[t][dir] < 0) ||
          (j+dx[t][dir] >= MAXN || j+dx[t][dir] < 0))){
            dir++;
            dir %= dx[t].size();
        }

        i += dy[t][dir],j += dx[t][dir];

    }
}
```

9.17 Random numbers in python (to create tests)

```
import random as r
r.random(0) #random float beetween 0 and 1
r.uniform(2.5, 100) #random float beetween 2.5 and 100
r.randrange(10) #random int beetween 0 and 10-1
r.choice(['win', 'lose', 'draw']) #Single random element from a sequence
r.shuffle(V) #random permutation of V

#random permutation of the numbers beetween 1 and n, with n radom
n = r.randrange(100)
def f(n):
     V=[]
     for i in range(n):
             V.append(i)
     r.shuffle(V)
     return V
```