Índice

1. Teoría de números

1.1. Funciones básicas

1.1.1. Función piso y techo

```
lli piso(lli a, lli b){
  if((a >= 0 && b > 0) || (a < 0 && b < 0)){
    return a / b;
  }else{
    if(a % b == 0) return a / b;
    else return a / b - 1;
  }
}

lli techo(lli a, lli b){
  if((a >= 0 && b > 0) || (a < 0 && b < 0)){
    if(a % b == 0) return a / b;
    else return a / b + 1;
  }else{
    return a / b;
  }
}
```

1.1.2. Exponenciación y multiplicación binaria

```
lli pow(lli b, lli e){
  lli ans = 1;
  while(e){
    if(e & 1) ans *= b;
    e >>= 1;
    b *= b;
  }
  return ans;
}

lli multMod(lli a, lli b, lli n){
  lli ans = 0;
  a %= n, b %= n;
  if(abs(b) > abs(a)) swap(a, b);
  if(b < 0){
    a *= -1, b *= -1;
  }
  while(b){
    if(b & 1) ans = (ans + a) % n;
    b >>= 1;
    a = (a + a) % n;
  }
  return ans;
}
```

1.1.3. Mínimo común múltiplo y máximo común divisor

```
lli gcd(lli a, lli b){
  lli r;
  while(b != 0) r = a % b, a = b, b = r;
  return a;
}

lli lcm(lli a, lli b){
  return b * (a / gcd(a, b));
}

lli gcd(vector<lli> & nums){
  lli ans = 0;
  for(lli & num : nums) ans = gcd(ans, num);
  return ans;
}

lli lcm(vector<lli> & nums){
  lli ans = 1;
  for(lli & num : nums) ans = lcm(ans, num);
  return ans;
}
```

1.1.4. Euclides extendido e inverso modular

```
lli extendedGcd(lli a, lli b, lli & s, lli & t){
  lli q, r0 = a, r1 = b, ri, s0 = 1, s1 = 0, si, t0 = 0, t1 =
  ↪ 1, ti;
```

```
  while(r1){
    q = r0 / r1;
    ri = r0 % r1, r0 = r1, r1 = ri;
    si = s0 - s1 * q, s0 = s1, s1 = si;
    ti = t0 - t1 * q, t0 = t1, t1 = ti;
  }
  s = s0, t = t0;
  return r0;
}

lli modularInverse(lli a, lli m){
  lli r0 = a, r1 = m, ri, s0 = 1, s1 = 0, si;
  while(r1){
    si = s0 - s1 * (r0 / r1), s0 = s1, s1 = si;
    ri = r0 % r1, r0 = r1, r1 = ri;
  }
  if(r0 < 0) s0 *= -1;
  if(s0 < 0) s0 += m;
  return s0;
}
```

1.1.5. Todos los inversos módulo p

```
//find all inverses (from 1 to p-1) modulo p
vector<lli> allInverses(lli p){
  vector<lli> ans(p);
  ans[1] = 1;
  for(lli i = 2; i < p; ++i)
    ans[i] = p - (p / i) * ans[p % i] % p;
  return ans;
}
```

1.1.6. Exponenciación binaria modular

```
lli powMod(lli b, lli e, lli m){
  lli ans = 1;
  b %= m;
  if(e < 0){
    b = modularInverse(b, m);
    e *= -1;
  }
  while(e){
    if(e & 1) ans = (ans * b) % m;
    e >>= 1;
    b = (b * b) % m;
  }
  return ans;
}
```

1.1.7. Teorema chino del residuo

```
pair<lli, lli> chinese(vector<lli> & a, vector<lli> & n){
  lli prod = 1, p, ans = 0;
  for(lli & ni : n) prod *= ni;
  for(int i = 0; i < a.size(); i++){
    p = prod / n[i];
    ans = (ans + (a[i] % n[i]) * modularInverse(p, n[i]) % prod
    ↪  * p) % prod;
  }
  if(ans < 0) ans += prod;
  return make_pair(ans, prod);
}
```

1.1.8. Coeficiente binomial

```
lli ncr(lli n, lli r){
  if(r < 0 || r > n) return 0;
  r = min(r, n - r);
  lli ans = 1;
  for(lli den = 1, num = n; den <= r; den++, num--){
    ans = ans * num / den;
  }
  return ans;
}
```

1.1.9. Fibonacci

```
//very fast fibonacci
inline void modula(lli & n){
  while(n < 0) n += mod;
  while(n >= mod) n -= mod;
}

array<lli, 2> mult(array<lli, 2> & A, array<lli, 2> & B){
  array<lli, 2> C;
  C[0] = A[0] * B[0] % mod;
  lli C2 = A[1] * B[1] % mod;
  C[1] = (A[0] + A[1]) * (B[0] + B[1]) % mod - (C[0] + C2);
  C[0] += C2;
  C[1] += C2;
  modula(C[0]), modula(C[1]);
  return C;
}

lli fibo(lli n){
  array<lli, 2> ans = {1, 0}, tmp = {0, 1};
  while(n){
    if(n & 1) ans = mult(ans, tmp);
    n >>= 1;
    if(n) tmp = mult(tmp, tmp);
  }
  return ans[1];
}
```

1.2. Cribas

1.2.1. Criba de divisores

```
vector<lli> divisorsSum;
vector<vector<lli>> divisors;
void divisorsSieve(lli n){
  divisorsSum.resize(n + 1, 0);
  divisors.resize(n + 1, vector<lli>());
  for(lli i = 1; i <= n; i++){
    for(lli j = i; j <= n; j += i){
      divisorsSum[j] += i;
      divisors[j].push_back(i);
    }
  }
}
```

1.2.2. Criba de primos

```
vector<lli> primes;
vector<bool> isPrime;
void primesSieve(lli n){
  isPrime.resize(n + 1, true);
  isPrime[0] = isPrime[1] = false;
  primes.push_back(2);
  for(lli i = 4; i <= n; i += 2){
    isPrime[i] = false;
  }
  for(lli i = 3; i <= n; i += 2){
    if(isPrime[i]){
      primes.push_back(i);
      for(lli j = i * i; j <= n; j += 2 * i){
        isPrime[j] = false;
      }
    }
  }
}
```

1.2.3. Criba de factor primo más pequeño

```
vector<lli> lowestPrime;
void lowestPrimeSieve(lli n){
  lowestPrime.resize(n + 1, 1);
  lowestPrime[0] = lowestPrime[1] = 0;
  for(lli i = 2; i <= n; i++) lowestPrime[i] = (i & 1 ? i : 2);
  lli limit = sqrt(n);
  for(lli i = 3; i <= limit; i += 2){
    if(lowestPrime[i] == i){
      for(lli j = i * i; j <= n; j += 2 * i){
        if(lowestPrime[j] == j) lowestPrime[j] = i;
```

```
      }
    }
  }
}
```

1.2.4. Criba de factores primos

```
vector<vector<lli>> primeFactors;
void primeFactorsSieve(lli n){
  primeFactors.resize(n + 1, vector<lli>());
  for(int i = 0; i < primes.size(); i++){
    lli p = primes[i];
    for(lli j = p; j <= n; j += p){
      primeFactors[j].push_back(p);
    }
  }
}
```

1.2.5. Criba de la función φ de Euler

```
vector<lli> Phi;
void phiSieve(lli n){
  Phi.resize(n + 1);
  for(lli i = 1; i <= n; i++) Phi[i] = i;
  for(lli i = 2; i <= n; i ++){
    if(Phi[i] == i){
      for(lli j = i; j <= n; j += i){
        Phi[j] -= Phi[j] / i;
      }
    }
  }
}
```

1.2.6. Triángulo de Pascal

```
vector<vector<lli>> Ncr;
void ncrSieve(lli n){
  Ncr.resize(n + 1, vector<lli>());
  Ncr[0] = {1};
  for(lli i = 1; i <= n; i++){
    Ncr[i].resize(i + 1);
    Ncr[i][0] = Ncr[i][i] = 1;
    for(lli j = 1; j <= i / 2; j++){
      Ncr[i][i - j] = Ncr[i][j] = Ncr[i - 1][j - 1] + Ncr[i -
      ↪ 1][j];
    }
  }
}
```

1.3. Factorización

1.3.1. Factorización de un número

```
vector<pair<lli, int>> factorize(lli n){
  vector<pair<lli, int>> f;
  for(lli & p : primes){
    if(p * p > n) break;
    int pot = 0;
    while(n % p == 0){
      pot++;
      n /= p;
    }
    if(pot) f.push_back(make_pair(p, pot));
  }
  if(n > 1) f.push_back(make_pair(n, 1));
  return f;
}
```

1.3.2. Potencia de un primo que divide a un factorial

```
lli potInFactorial(lli n, lli p){
  lli ans = 0;
  lli div = p;
  while(div <= n){
    ans += n / div;
    div *= p;
  }
  return ans;
```

```
}
```

1.3.3. Factorización de un factorial

```
vector<pair<lli, lli>> factorizeFactorial(lli n){
  vector<pair<lli, lli>> f;
  for(lli & p : primes){
    if(p > n) break;
    f.push_back(make_pair(p, potInFactorial(n, p)));
  }
  return f;
}
```

1.3.4. Factorización usando Pollard-Rho

```
bool isPrimeMillerRabin(lli n){
  if(n < 2) return false;
  if(n == 2) return true;
  lli d = n - 1, s = 0;
  while(!(d & 1)){
    d >>= 1;
    ++s;
  }
  for(int i = 0; i < 16; ++i){
    lli a = 1 + rand() % (n - 1);
    lli m = powMod(a, d, n);
    if(m == 1 || m == n - 1) goto exit;
    for(int k = 0; k < s - 1; ++k){
      m = m * m % n;
      if(m == n - 1) goto exit;
    }
    return false;
    exit:;
  }
  return true;
}

lli factorPollardRho(lli n){
  lli a = 1 + rand() % (n - 1);
  lli b = 1 + rand() % (n - 1);
  lli x = 2, y = 2, d = 1;
  while(d == 1 || d == -1){
    x = x * (x + b) % n + a;
    y = y * (y + b) % n + a;
    y = y * (y + b) % n + a;
    d = gcd(x - y, n);
  }
  return abs(d);
}

map<lli, int> fact;
void factorizePollardRho(lli n){
  while(n > 1 && !isPrimeMillerRabin(n)){
    lli f;
    do{
      f = factorPollardRho(n);
    }while(f == n);
    n /= f;
    factorizePollardRho(f);
    for(auto & it : fact){
      while(n % it.first == 0){
        n /= it.first;
        ++it.second;
      }
    }
  }
  if(n > 1) ++fact[n];
}
```

1.4. Funciones multiplicativas famosas

1.4.1. Función σ

```
//divisor power sum of n
//if pot=0 we get the number of divisors
//if pot=1 we get the sum of divisors
lli sigma(lli n, lli pot){
  lli ans = 1;
  vector<pair<lli, int>> f = factorize(n);
```

```cpp
  for(auto & factor : f){
    lli p = factor.first;
    int a = factor.second;
    if(pot){
      lli p_pot = pow(p, pot);
      ans *= (pow(p_pot, a + 1) - 1) / (p_pot - 1);
    }else{
      ans *= a + 1;
    }
  }
  return ans;
}
```

1.4.2. Función Ω

```cpp
//number of total primes with multiplicity dividing n
int Omega(lli n){
  int ans = 0;
  vector<pair<lli, int>> f = factorize(n);
  for(auto & factor : f){
    ans += factor.second;
  }
  return ans;
}
```

1.4.3. Función ω

```cpp
//number of distinct primes dividing n
int omega(lli n){
  int ans = 0;
  vector<pair<lli, int>> f = factorize(n);
  for(auto & factor : f){
    ++ans;
  }
  return ans;
}
```

1.4.4. Función φ de Euler

```cpp
//number of coprimes with n less than n
lli phi(lli n){
  lli ans = n;
  vector<pair<lli, int>> f = factorize(n);
  for(auto & factor : f){
    ans -= ans / factor.first;
  }
  return ans;
}
```

1.4.5. Función μ

```cpp
//1 if n is square-free with an even number of prime factors
//-1 if n is square-free with an odd number of prime factors
//0 is n has a square prime factor
int mu(lli n){
  int ans = 1;
  vector<pair<lli, int>> f = factorize(n);
  for(auto & factor : f){
    if(factor.second > 1) return 0;
    ans *= -1;
  }
  return ans;
}
```

1.5. Orden multiplicativo, raíces primitivas y raíces de la unidad

1.5.1. Función λ de Carmichael

```cpp
//the smallest positive integer k such that for
//every coprime x with n, x^k=1 mod n
lli carmichaelLambda(lli n){
  lli ans = 1;
  vector<pair<lli, int>> f = factorize(n);
  for(auto & factor : f){
    lli p = factor.first;
```

```
    int a = factor.second;
    lli tmp = pow(p, a);
    tmp -= tmp / p;
    if(a <= 2 || p >= 3) ans = lcm(ans, tmp);
    else ans = lcm(ans, tmp >> 1);
  }
  return ans;
}
```

1.5.2. Orden multiplicativo módulo m

```
// the smallest positive integer k such that x^k = 1 mod m
lli multiplicativeOrder(lli x, lli m){
  if(gcd(x, m) != 1) return -1;
  lli order = phi(m);
  vector<pair<lli, int>> f = factorize(order);
  for(auto & factor : f){
    lli p = factor.first;
    int a = factor.second;
    order /= pow(p, a);
    lli tmp = powMod(x, order, m);
    while(tmp != 1){
      tmp = powMod(tmp, p, m);
      order *= p;
    }
  }
  return order;
}
```

1.5.3. Número de raíces primitivas (generadores) módulo m

```
//number of generators modulo m
lli numberOfGenerators(lli m){
  lli phi_m = phi(m);
  lli lambda_m = carmichaelLambda(m);
  if(phi_m == lambda_m) return phi(phi_m);
  else return 0;
}
```

1.5.4. Test individual de raíz primitiva módulo m

```
//test if order(x, m) = phi(m), i.e., x is a generator for Z/mZ
bool testPrimitiveRoot(lli x, lli m){
  if(gcd(x, m) != 1) return false;
  lli order = phi(m);
  vector<pair<lli, int>> f = factorize(order);
  for(auto & factor : f){
    lli p = factor.first;
    if(powMod(x, order / p, m) == 1) return false;
  }
  return true;
}
```

1.5.5. Test individual de raíz k-ésima de la unidad módulo m

```
//test if x^k = 1 mod m and k is the smallest for such x, i.e.,
↪ x^(k/p) != 1 for every prime divisor of k
bool testPrimitiveKthRootUnity(lli x, lli k, lli m){
  if(powMod(x, k, m) != 1) return false;
  vector<pair<lli, int>> f = factorize(k);
  for(auto & factor : f){
    lli p = factor.first;
    if(powMod(x, k / p, m) == 1) return false;
  }
  return true;
}
```

1.5.6. Encontrar la primera raíz primitiva módulo m

```
lli findFirstGenerator(lli m){
  lli order = phi(m);
  if(order != carmichaelLambda(m)) return -1; //just an
  ↪ optimization, not required
  vector<pair<lli, int>> f = factorize(order);
  for(lli x = 1; x < m; x++){
    if(gcd(x, m) != 1) continue;
    bool test = true;
```

```
    for(auto & factor : f){
      lli p = factor.first;
      if(powMod(x, order / p, m) == 1){
        test = false;
        break;
      }
    }
    if(test) return x;
  }
  return -1;
}
```

1.5.7. Encontrar la primera raíz k-ésima de la unidad módulo m

```
lli findFirstPrimitiveKthRootUnity(lli k, lli m){
  if(carmichaelLambda(m) % k != 0) return -1; //just an
  ↪  optimization, not required
  vector<pair<lli, int>> f = factorize(k);
  for(lli x = 1; x < m; x++){
    if(powMod(x, k, m) != 1) continue;
    bool test = true;
    for(auto & factor : f){
      lli p = factor.first;
      if(powMod(x, k / p, m) == 1){
        test = false;
        break;
      }
    }
    if(test) return x;
  }
  return -1;
}
```

1.5.8. Logaritmo discreto

```
// a^x = b mod m, a and m coprime
pair<lli, lli> discreteLogarithm(lli a, lli b, lli m){
  if(gcd(a, m) != 1) return make_pair(-1, 0);
  lli order = multiplicativeOrder(a, m);
  lli n = sqrt(order) + 1;
  lli a_n = powMod(a, n, m);
  lli ans = 0;
  unordered_map<lli, lli> firstHalf;
  lli current = a_n;
  for(lli p = 1; p <= n; p++){
    firstHalf[current] = p;
    current = (current * a_n) % m;
  }
  current = b % m;
  for(lli q = 0; q <= n; q++){
    if(firstHalf.count(current)){
      lli p = firstHalf[current];
      lli x = n * p - q;
      return make_pair(x % order, order);
    }
    current = (current * a) % m;
  }
  return make_pair(-1, 0);
}
```

1.5.9. Raíz k-ésima discreta

```
// x^k = b mod m, m has at least one generator
vector<lli> discreteRoot(lli k, lli b, lli m){
  if(b % m == 0) return {0};
  lli g = findFirstGenerator(m);
  lli power = powMod(g, k, m);
  pair<lli, lli> y0 = discreteLogarithm(power, b, m);
  if(y0.first == -1) return {};
  lli phi_m = phi(m);
  lli d = gcd(k, phi_m);
  vector<lli> x(d);
  x[0] = powMod(g, y0.first, m);
  lli inc = powMod(g, phi_m / d, m);
  for(lli i = 1; i < d; i++){
    x[i] = x[i - 1] * inc % m;
  }
  sort(x.begin(), x.end());
```

```
  return x;
}
```

1.6. Particiones

1.6.1. Función P (particiones de un entero positivo)

```
lli mod = 1e9 + 7;

vector<lli> P;

//number of ways to write n as a sum of positive integers
lli partitionsP(int n){
  if(n < 0) return 0;
  if(P[n]) return P[n];
  int pos1 = 1, pos2 = 2, inc1 = 4, inc2 = 5;
  lli ans = 0;
  for(int k = 1; k <= n; k++){
    lli tmp = (n >= pos1 ? P[n - pos1] : 0) + (n >= pos2 ? P[n
    ↪  - pos2] : 0);
    if(k & 1){
      ans += tmp;
    }else{
      ans -= tmp;
    }
    if(n < pos2) break;
    pos1 += inc1, pos2 += inc2;
    inc1 += 3, inc2 += 3;
  }
  ans %= mod;
  if(ans < 0) ans += mod;
  return ans;
}

void calculateFunctionP(int n){
  P.resize(n + 1);
  P[0] = 1;
  for(int i = 1; i <= n; i++){
    P[i] = partitionsP(i);
  }
}
```

1.6.2. Función Q (particiones de un entero positivo en distintos sumandos)

```
vector<lli> Q;

bool isPerfectSquare(int n){
  int r = sqrt(n);
  return r * r == n;
}

int s(int n){
  int r = 1 + 24 * n;
  if(isPerfectSquare(r)){
    int j;
    r = sqrt(r);
    if((r + 1) % 6 == 0) j = (r + 1) / 6;
    else j = (r - 1) / 6;
    if(j & 1) return -1;
    else return 1;
  }else{
    return 0;
  }
}

//number of ways to write n as a sum of distinct positive
↪  integers
//number of ways to write n as a sum of odd positive integers
lli partitionsQ(int n){
  if(n < 0) return 0;
  if(Q[n]) return Q[n];
  int pos = 1, inc = 3;
  lli ans = 0;
  int limit = sqrt(n);
  for(int k = 1; k <= limit; k++){
    if(k & 1){
      ans += Q[n - pos];
    }else{
      ans -= Q[n - pos];
```

```
    }
    pos += inc;
    inc += 2;
  }
  ans <<= 1;
  ans += s(n);
  ans %= mod;
  if(ans < 0) ans += mod;
  return ans;
}

void calculateFunctionQ(int n){
  Q.resize(n + 1);
  Q[0] = 1;
  for(int i = 1; i <= n; i++){
    Q[i] = partitionsQ(i);
  }
}
```

1.6.3. Número de factorizaciones ordenadas

```
//number of ordered factorizations of n
lli orderedFactorizations(lli n){
  //skip the factorization if you already know the powers
  auto fact = factorize(n);
  int k = 0, q = 0;
  vector<int> powers(k + 1);
  for(auto & f : fact){
    powers[k + 1] = f.second;
    q += f.second;
    ++k;
  }
  vector<lli> prod(q + 1, 1);
  //we need Ncr until the max_power+Omega(n) row
  //module if needed
  for(int i = 0; i <= q; i++){
    for(int j = 1; j <= k; j++){
      prod[i] = prod[i] * Ncr[powers[j] + i][powers[j]];
    }
  }
  lli ans = 0;
  for(int j = 1; j <= q; j++){
    int alt = 1;
    for(int i = 0; i < j; i++){
      ans = ans + alt * Ncr[j][i] * prod[j - i - 1];
      alt *= -1;
    }
  }
  return ans;
}
```

1.6.4. Número de factorizaciones no ordenadas

```
//Number of unordered factorizations of n with
//largest part at most m
//Call unorderedFactorizations(n, n) to get all of them
//Add this to the main to speed up the map:
//mem.reserve(1024); mem.max_load_factor(0.25);
struct HASH{
  size_t operator()(const pair<int,int>&x)const{
    return hash<long long>()(((long long)x.first)^(((long
    ↪ long)x.second)<<32));
  }
};
unordered_map<pair<int, int>, lli, HASH> mem;
lli unorderedFactorizations(int m, int n){
  if(m == 1 && n == 1) return 1;
  if(m == 1) return 0;
  if(n == 1) return 1;
  if(mem.count({m, n})) return mem[{m, n}];
  lli ans = 0;
  int l = sqrt(n);
  for(int i = 1; i <= l; ++i){
    if(n % i == 0){
      lli a = i, b = n / i;
      if(a <= m) ans += unorderedFactorizations(a, b);
      if(a != b && b <= m) ans += unorderedFactorizations(b,
      ↪ a);
    }
  }
```

```
  return mem[{m, n}] = ans;
}
```

1.7. Otros

1.7.1. Cambio de base

```
string decimalToBaseB(lli n, lli b){
  string ans = "";
  lli digito;
  do{
    digito = n % b;
    if(0 <= digito && digito <= 9){
      ans = (char)(48 + digito) + ans;
    }else if(10 <= digito && digito <= 35){
      ans = (char)(55 + digito) + ans;
    }
    n /= b;
  }while(n != 0);
  return ans;
}

lli baseBtoDecimal(const string & n, lli b){
  lli ans = 0;
  for(const char & digito : n){
    if(48 <= digito && digito <= 57){
      ans = ans * b + (digito - 48);
    }else if(65 <= digito && digito <= 90){
      ans = ans * b + (digito - 55);
    }else if(97 <= digito && digito <= 122){
      ans = ans * b + (digito - 87);
    }
  }
  return ans;
}
```

1.7.2. Fracciones continuas

```
//continued fraction of (p+sqrt(n))/q, where p,n,q are positive
↪ integers
//returns a vector of terms and the length of the period,
//the periodic part is taken from the right of the array
pair<vector<lli>, int> ContinuedFraction(lli p, lli n, lli q){
  vector<lli> coef;
  lli r = sqrt(n);
  if(r * r == n){
    lli num = p + r;
    lli den = q;
    lli residue;
    while(den){
      residue = num % den;
      coef.push_back(num / den);
      num = den;
      den = residue;
    }
    return make_pair(coef, 0);
  }
  if((n - p * p) % q != 0){
    n *= q * q;
    p *= q;
    q *= q;
    r = sqrt(n);
  }
  lli a = (r + p) / q;
  coef.push_back(a);
  int period = 0;
  map<pair<lli, lli>, int> pairs;
  while(true){
    p = a * q - p;
    q = (n - p * p) / q;
    a = (r + p) / q;
    if(pairs.count(make_pair(p, q))){ //if p=0 and q=1, we can
    ↪ just ask if q==1 after inserting a
      period -= pairs[make_pair(p, q)];
      break;
    }
    coef.push_back(a);
```

```
    pairs[make_pair(p, q)] = period++;
  }
  return make_pair(coef, period);
}
```

1.7.3. Ecuación de Pell

```
//first solution (x, y) to the equation x^2-ny^2=1
pair<lli, lli> PellEquation(lli n){
  vector<lli> cf = ContinuedFraction(0, n, 1).first;
  lli num = 0, den = 1;
  int k = cf.size() - 1;
  for(int i = ((k & 1) ? (2 * k - 1) : (k - 1)); i >= 0; i--){
    lli tmp = den;
    int pos = i % k;
    if(pos == 0 && i != 0) pos = k;
    den = num + cf[pos] * den;
    num = tmp;
  }
  return make_pair(den, num);
}
```

2. Números racionales

2.1. Estructura fraccion

```
struct fraccion{
    lli num, den;
    fraccion(){
        num = 0, den = 1;
    }
    fraccion(lli x, lli y){
        if(y < 0)
            x *= -1, y *=-1;
        lli d = __gcd(abs(x), abs(y));
        num = x/d, den = y/d;
    }
    fraccion(lli v){
        num = v;
        den = 1;
    }
    fraccion operator+(const fraccion& f) const{
        lli d = __gcd(den, f.den);
        return fraccion(num*(f.den/d) + f.num*(den/d),
        ↪  den*(f.den/d));
    }
    fraccion operator-() const{
        return fraccion(-num, den);
    }
    fraccion operator-(const fraccion& f) const{
        return *this + (-f);
    }
    fraccion operator*(const fraccion& f) const{
        return fraccion(num*f.num, den*f.den);
    }
    fraccion operator/(const fraccion& f) const{
        return fraccion(num*f.den, den*f.num);
    }
    fraccion operator+=(const fraccion& f){
        *this = *this + f;
        return *this;
    }
    fraccion operator-=(const fraccion& f){
```

```
        *this = *this - f;
        return *this;
    }
    fraccion operator++(int xd){
        *this = *this + 1;
        return *this;
    }
    fraccion operator--(int xd){
        *this = *this - 1;
        return *this;
    }
    fraccion operator*=(const fraccion& f){
        *this = *this * f;
        return *this;
    }
    fraccion operator/=(const fraccion& f){
        *this = *this / f;
        return *this;
    }
    bool operator==(const fraccion& f) const{
        lli d = __gcd(den, f.den);
        return (num*(f.den/d) == (den/d)*f.num);
    }
    bool operator!=(const fraccion& f) const{
        lli d = __gcd(den, f.den);
        return (num*(f.den/d) != (den/d)*f.num);
    }
    bool operator >(const fraccion& f) const{
        lli d = __gcd(den, f.den);
        return (num*(f.den/d) > (den/d)*f.num);
    }
    bool operator <(const fraccion& f) const{
        lli d = __gcd(den, f.den);
        return (num*(f.den/d) < (den/d)*f.num);
    }
    bool operator >=(const fraccion& f) const{
        lli d = __gcd(den, f.den);
        return (num*(f.den/d) >= (den/d)*f.num);
    }
    bool operator <=(const fraccion& f) const{
        lli d = __gcd(den, f.den);
        return (num*(f.den/d) <= (den/d)*f.num);
    }
    fraccion inverso() const{
        return fraccion(den, num);
    }
    fraccion fabs() const{
        fraccion nueva;
        nueva.num = abs(num);
        nueva.den = den;
        return nueva;
    }
    double value() const{
      return (double)num / (double)den;
    }
    string str() const{
        stringstream ss;
        ss << num;
        if(den != 1) ss << "/" << den;
        return ss.str();
    }
};

ostream &operator<<(ostream &os, const fraccion & f) {
    return os << f.str();
}

istream &operator>>(istream &is, fraccion & f){
    lli num = 0, den = 1;
    string str;
    is >> str;
    size_t pos = str.find("/");
    if(pos == string::npos){
        istringstream(str) >> num;
    }else{
        istringstream(str.substr(0, pos)) >> num;
        istringstream(str.substr(pos + 1)) >> den;
    }
    f = fraccion(num, den);
    return is;
}
```

3. Álgebra lineal

3.1. Estructura matrix

```
template <typename entrada>
struct matrix{
  vector< vector<entrada> > A;
  int m, n;

  matrix(int _m, int _n){
    m = _m, n = _n;
    A.resize(m, vector<entrada>(n, 0));
  }

  vector<entrada> & operator[] (int i){
    return A[i];
  }

  void multiplicarFilaPorEscalar(int k, entrada c){
    for(int j = 0; j < n; j++) A[k][j] *= c;
  }

  void intercambiarFilas(int k, int l){
    swap(A[k], A[l]);
  }

  void sumaMultiploFilaAOtra(int k, int l, entrada c){
    for(int j = 0; j < n; j++) A[k][j] += c * A[l][j];
  }

  matrix operator+(const matrix & B) const{
    if(m == B.m && n == B.n){
      matrix<entrada> C(m, n);
      for(int i = 0; i < m; i++){
        for(int j = 0; j < n; j++){
          C[i][j] = A[i][j] + B.A[i][j];
        }
      }
      return C;
    }else{
      return *this;
    }
  }

  matrix operator+=(const matrix & M){
    *this = *this + M;
    return *this;
  }

  matrix operator-() const{
    matrix<entrada> C(m, n);
    for(int i = 0; i < m; i++){
      for(int j = 0; j < n; j++){
        C[i][j] = -A[i][j];
      }
    }
    return C;
  }

  matrix operator-(const matrix & B) const{
    return *this + (-B);
  }

  matrix operator-=(const matrix & M){
    *this = *this + (-M);
    return *this;
  }

  matrix operator*(const matrix & B) const{
    if(n == B.m){
      matrix<entrada> C(m, B.n);
      for(int i = 0; i < m; i++){
        for(int j = 0; j < B.n; j++){
          for(int k = 0; k < n; k++){
            C[i][j] += A[i][k] * B.A[k][j];
          }
        }
      }
      return C;
    }else{
      return *this;
```

```
  }
}

matrix operator*(const entrada & c) const{
  matrix<entrada> C(m, n);
  for(int i = 0; i < m; i++){
    for(int j = 0; j < n; j++){
      C[i][j] = A[i][j] * c;
    }
  }
  return C;
}

matrix operator*=(const matrix & M){
  *this = *this * M;
  return *this;
}

matrix operator*=(const entrada & c){
  *this = *this * c;
  return *this;
}

matrix operator^(lli b) const{
  matrix<entrada> ans = matrix<entrada>::identidad(n);
  matrix<entrada> A = *this;
  while(b){
    if(b & 1) ans *= A;
    b >>= 1;
    if(b) A *= A;
  }
  return ans;
}

matrix operator^=(lli n){
  *this = *this ^ n;
  return *this;
}

bool operator==(const matrix & B) const{
  if(m == B.m && n == B.n){
    for(int i = 0; i < m; i++){
      for(int j = 0; j < n; j++){
        if(A[i][j] != B.A[i][j]) return false;
      }
    }
    return true;
  }else{
    return false;
  }
}

bool operator!=(const matrix & B) const{
  return !(*this == B);
}
```

3.2. Gauss Jordan

```
//For every elemental operation that we apply to the matrix,
//we will call to callback(operation, source row, dest row,
↪  value).
//It returns the rank of the matrix, and modifies it
int gauss_jordan(bool full = true, bool makeOnes = true,
↪  function<void(int, int, int, entrada)>callback = NULL){
  int i = 0, j = 0;
  while(i < m && j < n){
    if(A[i][j] == 0){
      for(int f = i + 1; f < m; f++){
        if(A[f][j] != 0){
          intercambiarFilas(i, f);
          if(callback) callback(2, i, f, 0);
          break;
        }
      }
    }
    if(A[i][j] != 0){
      entrada inv_mult = A[i][j].inverso();
      if(makeOnes && A[i][j] != 1){
        multiplicarFilaPorEscalar(i, inv_mult);
        if(callback) callback(1, i, 0, inv_mult);
      }
```

```
      for(int f = (full ? 0 : (i + 1)); f < m; f++){
        if(f != i && A[f][j] != 0){
          entrada inv_adit = -A[f][j];
          if(!makeOnes) inv_adit *= inv_mult;
          sumaMultiploFilaAOtra(f, i, inv_adit);
          if(callback) callback(3, f, i, inv_adit);
        }
      }
      i++;
    }
    j++;
  }
  return i;
}

void eliminacion_gaussiana(){
  gauss_jordan(false);
}
```

3.3. Matriz inversa

```
static matrix identidad(int n){
  matrix<entrada> id(n, n);
  for(int i = 0; i < n; i++){
    id[i][i] = 1;
  }
  return id;
}

matrix<entrada> inversa(){
  if(m == n){
    matrix<entrada> tmp = *this;
    matrix<entrada> inv = matrix<entrada>::identidad(n);
    auto callback = [&](int op, int a, int b, entrada e){
      if(op == 1){
        inv.multiplicarFilaPorEscalar(a, e);
      }else if(op == 2){
        inv.intercambiarFilas(a, b);
      }else if(op == 3){
        inv.sumaMultiploFilaAOtra(a, b, e);
      }
    };
    if(tmp.gauss_jordan(true, true, callback) == n){
      return inv;
    }else{
      return *this;
    }
  }else{
    return *this;
  }
}
```

3.4. Transpuesta

```
matrix<entrada> transpuesta(){
  matrix<entrada> T(n, m);
  for(int i = 0; i < m; i++){
    for(int j = 0; j < n; j++){
      T[j][i] = A[i][j];
    }
  }
  return T;
}
```

3.5. Traza

```
entrada traza(){
  entrada sum = 0;
  for(int i = 0; i < min(m, n); i++){
    sum += A[i][i];
  }
  return sum;
}
```

3.6. Determinante

```
entrada determinante(){
  if(m == n){
```

```
    matrix<entrada> tmp = *this;
    entrada det = 1;
    auto callback = [&](int op, int a, int b, entrada e){
      if(op == 1){
        det /= e;
      }else if(op == 2){
        det *= -1;
      }
    };
    if(tmp.gauss_jordan(false, true, callback) != n) det = 0;
    return det;
  }else{
    return 0;
  }
}
```

3.7. Matriz de cofactores y adjunta

```
matrix<entrada> menor(int x, int y){
  matrix<entrada> M(0, 0);
  for(int i = 0; i < m; i++){
    if(i != x){
      M.A.push_back(vector<entrada>());
      for(int j = 0; j < n; j++){
        if(j != y){
          M.A.back().push_back(A[i][j]);
        }
      }
    }
  }
  M.m = m - 1;
  M.n = n - 1;
  return M;
}

entrada cofactor(int x, int y){
  entrada ans = menor(x, y).determinante();
  if((x + y) % 2 == 1) ans *= -1;
  return ans;
}
```

```
matrix<entrada> cofactores(){
  matrix<entrada> C(m, n);
  for(int i = 0; i < m; i++){
    for(int j = 0; j < n; j++){
      C[i][j] = cofactor(i, j);
    }
  }
  return C;
}

matrix<entrada> adjunta(){
  return cofactores().transpuesta();
}
```

3.8. Factorización $PA = LU$

```
vector< matrix<entrada> > PA_LU(){
  matrix<entrada> U = *this;
  matrix<entrada> L = matrix<entrada>::identidad(n);
  matrix<entrada> P = matrix<entrada>::identidad(n);
  auto callback = [&](int op, int a, int b, entrada e){
    if(op == 2){
      L.intercambiarFilas(a, b);
      P.intercambiarFilas(a, b);
      L.A[a][a] = L.A[b][b] = 1;
      L.A[a][a + 1] = L.A[b][b - 1] = 0;
    }else if(op == 3){
      L.A[a][b] = -e;
    }
  };
  U.gauss_jordan(false, false, callback);
  return {P, L, U};
}
```

3.9. Polinomio característico

```
vector<entrada> polinomio(){
  matrix<entrada> M(n, n);
```

```
    vector<entrada> coef(n + 1);
    matrix<entrada> I = matrix<entrada>::identidad(n);
    coef[n] = 1;
    for(int i = 1; i <= n; i++){
      M = (*this) * M + I * coef[n - i + 1];
      coef[n - i] = -((*this) * M).traza() / i;
    }
    return coef;
  }
```

3.10. Gram-Schmidt

```
  matrix<entrada> gram_schmidt(){ //los vectores son las filas
  ↪  de la matriz
    matrix<entrada> B = (*this) * (*this).transpuesta();
    matrix<entrada> ans = *this;
    auto callback = [&](int op, int a, int b, entrada e){
      if(op == 1){
        ans.multiplicarFilaPorEscalar(a, e);
      }else if(op == 2){
        ans.intercambiarFilas(a, b);
      }else if(op == 3){
        ans.sumaMultiploFilaAOtra(a, b, e);
      }
    };
    B.gauss_jordan(false, false, callback);
    return ans;
  }
```

3.11. Recurrencias lineales

```
void multByOne(lli *polynomial, lli *original, int degree){
  lli first = polynomial[degree - 1];
  for(int i = degree - 1; i >= 0; --i){
    polynomial[i] = first * original[i];
    if(i > 0) polynomial[i] += polynomial[i - 1];
  }
  for(int i = 0; i < degree; ++i) polynomial[i] %= mod;
}

lli *mult(lli *P, lli *Q, lli **residues, int degree){
  lli *R = new lli[degree]();
  lli *S = new lli[degree - 1]();
  for(int i = 0; i < degree; i++){
    for(int j = 0; j < degree; j++){
      if(i + j < degree) R[i + j] += P[i] * Q[j];
      else S[i + j - degree] += P[i] * Q[j];
    }
  }
  for(int i = 0; i < degree - 1; i++) S[i] %= mod;
  for(int i = 0; i < degree - 1; i++){
    for(int j = 0; j < degree; j++)
      R[j] += S[i] * residues[i][j];
  }
  for(int i = 0; i < degree; i++) R[i] %= mod;
  return R;
}

lli **getResidues(lli *charPoly, int degree){
  lli **residues = new lli*[degree - 1];
  lli *current = new lli[degree];
  copy(charPoly, charPoly + degree, current);
  for(int i = 0; i < degree - 1; i++){
    residues[i] = new lli[degree];
    copy(current, current + degree, residues[i]);
    if(i != degree - 2) multByOne(current, charPoly, degree);
  }
  return residues;
}

//Solves a linear recurrence relation of degree d of the form
//F(n) = a(d-1)*F(n-1) + a(d-2)*F(n-2) + ... + a(1)*F(n-(d-1))
↪  + a(0)*F(n-d)
//with initial values F(0), F(1), ..., F(d-1)
//It finds the nth term of the recurrence, F(n)
//The values of a[0,...,d) are in the array charPoly[]
lli solveRecurrence(lli *charPoly, lli *initValues, int degree,
↪  lli n){
  lli **residues = getResidues(charPoly, degree);
  lli *tmp = new lli[degree]();
```

```
  lli *ans = new lli[degree]();
  ans[0] = 1;
  if(degree > 1) tmp[1] = 1;
  else tmp[0] = charPoly[0];
  while(n){
    if(n & 1) ans = mult(ans, tmp, residues, degree);
    n >>= 1;
    if(n) tmp = mult(tmp, tmp, residues, degree);
  }
  lli nValue = 0;
  for(int i = 0; i < degree; i++) nValue += ans[i] *
  ↪  initValues[i];
  return nValue % mod;
}
```

4. FFT

4.1. Funciones previas

```
typedef complex<double> comp;
typedef long long int lli;
double PI = acos(-1.0);

int nearestPowerOfTwo(int n){
  int ans = 1;
  while(ans < n) ans <<= 1;
  return ans;
}

bool isZero(comp z){
  return abs(z.real()) < 1e-3;
}

template<typename T>
void swapPositions(vector<T> & X){
  int n = X.size();
  int bit;
  for (int i = 1, j = 0; i < n; ++i) {
    bit = n >> 1;
    while(j & bit){
      j ^= bit;
      bit >>= 1;
    }
    j ^= bit;
    if (i < j){
      swap (X[i], X[j]);
    }
  }
}
```

4.2. FFT con raíces de la unidad complejas

```
void fft(vector<comp> & X, int inv){
  int n = X.size();
    swapPositions<comp>(X);
```

```
    int len, len2, i, j;
    double ang;
    comp t, u, v;
    vector<comp> wlen_pw(n >> 1);
    wlen_pw[0] = 1;
    for(len = 2; len <= n; len <<= 1) {
        ang = inv == -1 ? -2 * PI / len : 2 * PI / len;
        len2 = len >> 1;
        comp wlen(cos(ang), sin(ang));
        for(i = 1; i < len2; ++i){
            wlen_pw[i] = wlen_pw[i - 1] * wlen;
        }
        for(i = 0; i < n; i += len) {
            for(j = 0; j < len2; ++j) {
                t = X[i + j + len2] * wlen_pw[j];
                X[i + j + len2] = X[i + j] - t;
                X[i + j] += t;
            }
        }
    }
    if(inv == -1){
        for(i = 0; i < n; ++i){
            X[i] /= n;
        }
    }
}
```

4.3. FFT con raíces de la unidad discretas (NTT)

```
const int p = 7340033;
const int root = 5;
const int root_1 = 4404020;
const int root_pw = 1 << 20;

int inverse(int a, int n){
    int r0 = a, r1 = n, ri, s0 = 1, s1 = 0, si;
    while(r1){
        si = s0 - s1 * (r0 / r1), s0 = s1, s1 = si;
        ri = r0 % r1, r0 = r1, r1 = ri;
    }
    if(s0 < 0) s0 += n;
    return s0;
}

void ntt(vector<int> & X, int inv) {
  int n = X.size();
  swapPositions<int>(X);
  int len, len2, wlen, i, j, u, v, w;
  for (len = 2; len <= n; len <<= 1) {
    len2 = len >> 1;
    wlen = (inv == -1) ? root_1 : root;
    for (i = len; i < root_pw; i <<= 1){
      wlen = wlen * 1ll * wlen % p;
    }
    for (i = 0; i < n; i += len) {
      w = 1;
      for (j = 0; j < len2; ++j) {
        u = X[i + j], v = X[i + j + len2] * 1ll * w % p;
        X[i + j] = u + v < p ? u + v : u + v - p;
        X[i + j + len2] = u - v < 0 ? u - v + p : u - v;
        w = w * 1ll * wlen % p;
      }
    }
  }
  if (inv == -1) {
    int nrev = inverse(n, p);
    for (i = 0; i < n; ++i){
      X[i] = X[i] * 1ll * nrev % p;
    }
  }
}
```

4.3.1. Otros valores para escoger la raíz y el módulo

Raíz n-ésima de la unidad (ω)	ω^{-1}	Tamaño máximo del arreglo (n)	Módulo p
15	30584	2^{14}	$4 \times 2^{14} + 1 = 65537$
9	7282	2^{15}	$2 \times 2^{15} + 1 = 65537$
3	21846	2^{16}	$1 \times 2^{16} + 1 = 65537$
8	688129	2^{17}	$6 \times 2^{17} + 1 = 786433$
5	471860	2^{18}	$3 \times 2^{18} + 1 = 786433$
12	3364182	2^{19}	$11 \times 2^{19} + 1 = 5767169$
5	**4404020**	$\mathbf{2^{20}}$	$7 \times 2^{20} + 1 = \mathbf{7340033}$
38	21247462	2^{21}	$11 \times 2^{21} + 1 = 23068673$
21	49932191	2^{22}	$25 \times 2^{22} + 1 = 104857601$
4	125829121	2^{23}	$20 \times 2^{23} + 1 = 167772161$
31	**128805723**	$\mathbf{2^{23}}$	$119 \times 2^{23} + 1 = \mathbf{998244353}$
2	83886081	2^{24}	$10 \times 2^{24} + 1 = 167772161$
17	29606852	2^{25}	$5 \times 2^{25} + 1 = 167772161$
30	15658735	2^{26}	$7 \times 2^{26} + 1 = 469762049$
137	749463956	2^{27}	$15 \times 2^{27} + 1 = 2013265921$

4.4. Aplicaciones

4.4.1. Multiplicación de polinomios

```
void multiplyPolynomials(vector<comp> & A, vector<comp> & B){
  int degree = A.size() + B.size() - 2;
  int size = nearestPowerOfTwo(degree + 1);
  A.resize(size);
  B.resize(size);
  fft(A, 1);
  fft(B, 1);
  for(int i = 0; i < size; i++){
    A[i] *= B[i];
  }
  fft(A, -1);
  A.resize(degree + 1);
}

void multiplyPolynomials(vector<int> & A, vector<int> & B){
  int degree = A.size() + B.size() - 2;
  int size = nearestPowerOfTwo(degree + 1);
  A.resize(size);
  B.resize(size);
  ntt(A, 1);
  ntt(B, 1);
  for(int i = 0; i < size; i++){
    A[i] = A[i] * 1ll * B[i] % p;
  }
  ntt(A, -1);
  A.resize(degree + 1);
}
```

4.4.2. Multiplicación de números enteros grandes

```
string multiplyNumbers(const string & a, const string & b){
  int sgn = 1;
  int pos1 = 0, pos2 = 0;
  while(pos1 < a.size() && (a[pos1] < '1' || a[pos1] > '9')){
    if(a[pos1] == '-') sgn *= -1;
    ++pos1;
  }
  while(pos2 < b.size() && (b[pos2] < '1' || b[pos2] > '9')){
    if(b[pos2] == '-') sgn *= -1;
    ++pos2;
  }
  vector<int> X(a.size() - pos1), Y(b.size() - pos2);
  if(X.empty() || Y.empty()) return "0";
  for(int i = pos1, j = X.size() - 1; i < a.size(); ++i){
    X[j--] = a[i] - '0';
  }
  for(int i = pos2, j = Y.size() - 1; i < b.size(); ++i){
    Y[j--] = b[i] - '0';
  }
  multiplyPolynomials(X, Y);
  stringstream ss;
  if(sgn == -1) ss << "-";
```

```
  int carry = 0;
  for(int i = 0; i < X.size(); ++i){
    X[i] += carry;
    carry = X[i] / 10;
    X[i] %= 10;
  }
  while(carry){
    X.push_back(carry % 10);
    carry /= 10;
  }
  for(int i = X.size() - 1; i >= 0; --i){
    ss << X[i];
  }
  return ss.str();
}
```

5. Geometría

5.1. Estructura point

```
double eps = 1e-8;
# define M_PI 3.14159265358979323846
# define M_E 2.71828182845904523536

struct point{
  double x, y;

  point(){
    x = y = 0;
  }
  point(double x, double y){
    this->x = x, this->y = y;
  }

  point operator+(const point & p) const{
    return point(x + p.x, y + p.y);
  }
  point operator-(const point & p) const{
    return point(x - p.x, y - p.y);
  }
  point operator*(const double & k) const{
    return point(x * k, y * k);
  }
  point operator/(const double & k) const{
    return point(x / k, y / k);
  }

  point rotate(const double angle) const{
    return point(x * cos(angle) - y * sin(angle), x *
    ↪  sin(angle) + y * cos(angle));
  }
  point rotate(const double angle, const point & p){
    return p + ((*this) - p).rotate(angle);
  }

  double dot(const point & p) const{
    return x * p.x + y * p.y;
```

```cpp
  }
  double length() const{
    return hypot(x, y);
  }
  double cross(const point & p) const{
    return x * p.y - y * p.x;
  }

  point normalize() const{
    return (*this) / length();
  }

  point projection(const point & p) const{
    return (*this) * p.dot(*this) / dot(*this);
  }
  point normal(const point & p) const{
    return p - projection(p);
  }

  bool operator==(const point & p) const{
    return abs(x - p.x) < eps && abs(y - p.y) < eps;
  }
  bool operator!=(const point & p) const{
    return !(*this == p);
  }
  bool operator<(const point & p) const{
    if(abs(x - p.x) < eps){
      return y < p.y;
    }else{
      return x < p.x;
    }
  }
  bool operator>(const point & p) const{
    if(abs(x - p.x) < eps){
      return y > p.y;
    }else{
      return x > p.x;
    }
  }
};
istream &operator>>(istream &is, point & P){
  point p;
    is >> p.x >> p.y;
    P = p;
    return is;
}

ostream &operator<<(ostream &os, const point & p) {
    return os << fixed << setprecision(8) << p.x << " " << p.y;
}

int sgn(double x){
  if(abs(x) < eps){
    return 0;
  }else if(x > 0){
    return 1;
  }else{
    return -1;
  }
}
```

5.2. Verificar si un punto pertenece a una línea o segmento

```cpp
bool pointInLine(point & a, point & b, point & p){
  //line ab, point p
  return abs((p - a).cross(b - a)) < eps;
}

bool pointInSegment(point a, point b, point & p){
  //segment ab, point p
  if(a > b) swap(a, b);
  return pointInLine(a, b, p) && !(p < a || p > b);
}
```

5.3. Intersección de líneas

```
int intersectLinesInfo(point & a, point & b, point & c, point &
↪ d){
  //line ab, line cd
  point v1 = b - a, v2 = d - c;
  double det = v1.cross(v2);
  if(abs(det) < eps){
    if(abs((c - a).cross(v1)) < eps){
      return -1; //infinity points
    }else{
      return 0; //no points
    }
  }else{
    return 1; //single point
  }
}

point intersectLines(point & a, point & b, point & c, point &
↪ d){
  //assuming that they intersect
  point v1 = b - a, v2 = d - c;
  double det = v1.cross(v2);
  return a + v1 * ((c - a).cross(v2) / det);
}
```

5.4. Intersección de segmentos

```
int intersectSegmentsInfo(point & a, point & b, point & c,
↪ point & d){
  //segment ab, segment cd
  point v1 = b - a, v2 = d - c;
  int t = sgn(v1.cross(c - a)), u = sgn(v1.cross(d - a));
  if(t == u){
    if(t == 0){
      if(pointInSegment(a, b, c) || pointInSegment(a, b, d) ||
      ↪ pointInSegment(c, d, a) || pointInSegment(c, d, b)){
        return -1; //infinity points
      }else{
        return 0; //no point
      }
    }else{
      return 0; //no point
    }
  }else{
    return sgn(v2.cross(a - c)) != sgn(v2.cross(b - c)); //1:
    ↪ single point, 0: no point
  }
}
```

5.5. Distancia punto-recta

```
double distancePointLine(point & a, point & v, point & p){
  //line: a + tv, point p
  return abs(v.cross(p - a)) / v.length();
}
```

5.6. Perímetro y área de un polígono

```
double perimeter(vector<point> & points){
  int n = points.size();
  double ans = 0;
  for(int i = 0; i < n; i++){
    ans += (points[i] - points[(i + 1) % n]).length();
  }
  return ans;
}

double area(vector<point> & points){
  int n = points.size();
  double ans = 0;
  for(int i = 0; i < n; i++){
    ans += points[i].cross(points[(i + 1) % n]);
  }
  return abs(ans / 2);
}
```

5.7. Envolvente convexa (convex hull) de un polígono

```
vector<point> convexHull(vector<point> points){
  sort(points.begin(), points.end());
  vector<point> L, U;
  for(int i = 0; i < points.size(); i++){
    while(L.size() >= 2 && (L[L.size() - 2] -
    ↪  points[i]).cross(L[L.size() - 1] - points[i]) <= 0){
      L.pop_back();
    }
    L.push_back(points[i]);
  }
  for(int i = points.size() - 1; i >= 0; i--){
    while(U.size() >= 2 && (U[U.size() - 2] -
    ↪  points[i]).cross(U[U.size() - 1] - points[i]) <= 0){
      U.pop_back();
    }
    U.push_back(points[i]);
  }
  L.pop_back();
  U.pop_back();
  L.insert(L.end(), U.begin(), U.end());
  return L;
}
```

5.8. Verificar si un punto pertenece al perímetro de un polígono

```
bool pointInPerimeter(vector<point> & points, point & p){
  int n = points.size();
  for(int i = 0; i < n; i++){
    if(pointInSegment(points[i], points[(i + 1) % n], p)){
      return true;
    }
  }
  return false;
}
```

5.9. Verificar si un punto pertenece a un polígono

```
int pointInPolygon(vector<point> & points, point & p){
  if(pointInPerimeter(points, p)){
    return -1; //point in the perimeter
  }
  point bottomLeft = (*min_element(points.begin(),
  ↪  points.end())) - point(M_E, M_PI);
  int n = points.size();
  int rays = 0;
  for(int i = 0; i < n; i++){
    rays += (intersectSegmentsInfo(p, bottomLeft, points[i],
    ↪  points[(i + 1) % n]) == 1 ? 1 : 0);
  }
  return rays & 1; //0: point outside, 1: point inside
}
```

5.10. Par de puntos más cercanos

```
bool comp1(const point & a, const point & b){
  return a.y < b.y;
}
pair<point, point> closestPairOfPoints(vector<point> points){
  sort(points.begin(), points.end(), comp1);
  set<point> S;
  double ans = 1e9;
  point p, q;
  int pos = 0;
  for(int i = 0; i < points.size(); ++i){
    while(pos < i && abs(points[i].y - points[pos].y) >= ans){
      S.erase(points[pos++]);
    }
    auto lower = S.lower_bound({-1e9, points[i].x - ans -
    ↪  eps});
    auto upper = S.upper_bound({-1e9, points[i].x + ans +
    ↪  eps});
    for(auto it = lower; it != upper; ++it){
      double d = (points[i] - *it).length();
      if(d < ans){
        ans = d;
```

```
      p = points[i];
      q = *it;
    }
  }
  S.insert(points[i]);
 }
 return {p, q};
}
```

6. Grafos

6.1. Estructura disjointSet

```
struct disjointSet{
  int N;
  vector<short int> rank;
  vector<int> parent;

  disjointSet(int N){
    this->N = N;
    parent.resize(N);
    rank.resize(N);
  }

  void makeSet(int v){
    parent[v] = v;
  }

  int findSet(int v){
    if(v == parent[v]) return v;
    return parent[v] = findSet(parent[v]);
  }

  void unionSet(int a, int b){
    a = findSet(a);
    b = findSet(b);
    if(a == b) return;
    if(rank[a] < rank[b]){
      parent[a] = b;
    }else{
      parent[b] = a;
      if(rank[a] == rank[b]){
        ++rank[a];
      }
    }
  }
};
```

6.2. Estructura edge

```
struct edge{
  int source, dest, cost;
  edge(){
    this->source = this->dest = this->cost = 0;
  }
  edge(int dest, int cost){
    this->dest = dest;
    this->cost = cost;
  }
  edge(int source, int dest, int cost){
    this->source = source;
    this->dest = dest;
    this->cost = cost;
  }
  bool operator==(const edge & b) const{
    return source == b.source && dest == b.dest && cost ==
    ↪  b.cost;
  }
  bool operator<(const edge & b) const{
    return cost < b.cost;
  }
  bool operator>(const edge & b) const{
    return cost > b.cost;
  }
};
```

6.3. Estructura path

```
struct path{
  int cost = inf;
  vector<int> vertices;
  int size = 1;
  int previous = -1;
};
```

6.4. Estructura graph

```
struct graph{
  vector<vector<edge>> adjList;
  vector<vector<bool>> adjMatrix;
  vector<vector<int>> costMatrix;
  vector<edge> edges;
  int V = 0;
  bool dir = false;

  graph(int n, bool dirigido){
    V = n;
    dir = dirigido;
    adjList.resize(V, vector<edge>());
    edges.resize(V);
    adjMatrix.resize(V, vector<bool>(V, false));
    costMatrix.resize(V, vector<int>(V, inf));
    for(int i = 0; i < V; i++)
      costMatrix[i][i] = 0;
  }

  void add(int source, int dest, int cost){
    adjList[source].push_back(edge(source, dest, cost));
    edges.push_back(edge(source, dest, cost));
    adjMatrix[source][dest] = true;
    costMatrix[source][dest] = cost;
    if(!dir){
      adjList[dest].push_back(edge(dest, source, cost));
      adjMatrix[dest][source] = true;
      costMatrix[dest][source] = cost;
    }
  }

  void buildPaths(vector<path> & paths){
    for(int i = 0; i < V; i++){
      int actual = i;
      for(int j = 0; j < paths[i].size; j++){
        paths[i].vertices.push_back(actual);
        actual = paths[actual].previous;
      }
```

```
      reverse(paths[i].vertices.begin(),
      ↪  paths[i].vertices.end());
    }
  }
```

6.5. Dijkstra con reconstrucción del camino más corto con menos vértices

```
vector<path> dijkstra(int start){
  priority_queue<edge, vector<edge>, greater<edge>> cola;
  vector<path> paths(V, path());
  vector<bool> relaxed(V, false);
  cola.push(edge(start, 0));
  paths[start].cost = 0;
  relaxed[start] = true;
  while(!cola.empty()){
    int u = cola.top().dest; cola.pop();
    relaxed[u] = true;
    for(edge & current : adjList[u]){
      int v = current.dest;
      if(relaxed[v]) continue;
      int nuevo = paths[u].cost + current.cost;
      if(nuevo == paths[v].cost && paths[u].size + 1 <
      ↪  paths[v].size){
        paths[v].previous = u;
        paths[v].size = paths[u].size + 1;
      }else if(nuevo < paths[v].cost){
        paths[v].previous = u;
        paths[v].size = paths[u].size + 1;
        cola.push(edge(v, nuevo));
        paths[v].cost = nuevo;
      }
    }
  }
  buildPaths(paths);
  return paths;
}
```

6.6. Bellman Ford con reconstrucción del camino más corto con menos vértices

```
vector<path> bellmanFord(int start){
  vector<path> paths(V, path());
  vector<int> processed(V);
  vector<bool> inQueue(V, false);
  queue<int> Q;
  paths[start].cost = 0;
  Q.push(start);
  while(!Q.empty()){
    int u = Q.front(); Q.pop(); inQueue[u] = false;
    if(paths[u].cost == inf) continue;
    ++processed[u];
    if(processed[u] == V){
      cout << "Negative cycle\n";
      return {};
    }
    for(edge & current : adjList[u]){
      int v = current.dest;
      int nuevo = paths[u].cost + current.cost;
      if(nuevo == paths[v].cost && paths[u].size + 1 <
      ↪  paths[v].size){
        paths[v].previous = u;
        paths[v].size = paths[u].size + 1;
      }else if(nuevo < paths[v].cost){
        if(!inQueue[v]){
          Q.push(v);
          inQueue[v] = true;
        }
        paths[v].previous = u;
        paths[v].size = paths[u].size + 1;
        paths[v].cost = nuevo;
      }
    }
  }
  buildPaths(paths);
  return paths;
}
```

6.7. Floyd

```
vector<vector<int>> floyd(){
  vector<vector<int>> tmp = costMatrix;
  for(int k = 0; k < V; ++k)
    for(int i = 0; i < V; ++i)
      for(int j = 0; j < V; ++j)
        if(tmp[i][k] != inf && tmp[k][j] != inf)
          tmp[i][j] = min(tmp[i][j], tmp[i][k] + tmp[k][j]);
  return tmp;
}
```

6.8. Cerradura transitiva $O(V^3)$

```
vector<vector<bool>> transitiveClosure(){
  vector<vector<bool>> tmp = adjMatrix;
  for(int k = 0; k < V; ++k)
    for(int i = 0; i < V; ++i)
      for(int j = 0; j < V; ++j)
        tmp[i][j] = tmp[i][j] || (tmp[i][k] && tmp[k][j]);
  return tmp;
}
```

6.9. Cerradura transitiva $O(V^2)$

```
void DFSClosure(int start, int u, vector<vector<bool>> &
↪  tmp){
  for(edge & current : adjList[u]){
    int v = current.dest;
    if(!tmp[start][v]){
      tmp[start][v] = true;
      DFSClosure(start, v, tmp);
    }
  }
}

vector<vector<bool>> transitiveClosureDFS(){
  vector<vector<bool>> tmp(V, vector<bool>(V, false));
  for(int u = 0; u < V; u++)
    DFSClosure(u, u, tmp);
  return tmp;
}
```

6.10. Verificar si el grafo es bipartito

```
bool isBipartite(){
  vector<int> side(V, -1);
  queue<int> q;
  for (int st = 0; st < V; ++st) {
    if(side[st] != -1) continue;
    q.push(st);
    side[st] = 0;
    while (!q.empty()) {
      int u = q.front();
      q.pop();
      for (edge & current : adjList[u]) {
        int v = current.dest;
        if (side[v] == -1) {
          side[v] = side[u] ^ 1;
          q.push(v);
        } else {
          if(side[v] == side[u]) return false;
        }
      }
    }
  }
  return true;
}
```

6.11. Orden topológico

```
vector<int> topologicalSort(){
  vector<int> order;
  int visited = 0;
  vector<int> indegree(V);
  for(auto & node : adjList){
    for(edge & current : node){
      int v = current.dest;
```

```
      ++indegree[v];
    }
  }
  queue<int> Q;
  for(int i = 0; i < V; ++i){
    if(indegree[i] == 0) Q.push(i);
  }
  while(!Q.empty()){
    int source = Q.front();
    Q.pop();
    order.push_back(source);
    ++visited;
    for(edge & current : adjList[source]){
      int v = current.dest;
      --indegree[v];
      if(indegree[v] == 0) Q.push(v);
    }
  }
  if(visited == V) return order;
  else return {};
}
```

6.12. Detectar ciclos

```
bool DFSCycle(int u, int parent, vector<int> & color){
  color[u] = 1;
  for(edge & current : adjList[u]){
    int v = current.dest;
    if(color[v] == 0)
      return DFSCycle(v, u, color);
    else if(color[v] == 1 && (dir || v != parent))
      return true;
  }
  color[u] = 2;
  return false;
}

bool hasCycle(){
  vector<int> color(V);
  for(int u = 0; u < V; ++u)
    if(color[u] == 0 && DFSCycle(u, -1, color))
      return true;
  return false;
}
```

6.13. Puentes y puntos de articulación

```
int articulationBridges(int u, int p, vector<int> & low,
↪ vector<int> & label, int & time, vector<bool> & points,
↪ vector<edge> & bridges){
  label[u] = low[u] = ++time;
  int hijos = 0, ret = 0;
  for(edge & current : adjList[u]){
    int v = current.dest;
    if(v == p && !ret++) continue;
    if(!label[v]){
      ++hijos;
      articulationBridges(v, u, low, label, time, points,
      ↪ bridges);
      if(label[u] <= low[v])
        points[u] = true;
      else if(label[u] < low[v])
        bridges.push_back(current);
      low[u] = min(low[u], low[v]);
    }
    low[u] = min(low[u], label[v]);
  }
  return hijos;
}

pair<vector<bool>, vector<edge>> articulationBridges(){
  vector<int> low(V), label(V);
  vector<bool> points(V);
  vector<edge> bridges;
  int time = 0;
  for(int u = 0; u < V; ++u)
    if(!label[u])
      points[u] = articulationBridges(u, -1, low, label,
      ↪ time, points, bridges) > 1;
  return make_pair(points, bridges);
```

```
}
```

6.14. Componentes fuertemente conexas

```
void scc(int u, vector<int> & low, vector<int> & label, int &
↪ time, vector<vector<int>> & ans, stack<int> & S){
  label[u] = low[u] = ++time;
  S.push(u);
  for(edge & current : adjList[u]){
    int v = current.dest;
    if(!label[v]) scc(v, low, label, time, ans, S);
    low[u] = min(low[u], low[v]);
  }
  if(label[u] == low[u]){
    vector<int> comp;
    while(S.top() != u){
      comp.push_back(S.top());
      low[S.top()] = V + 1;
      S.pop();
    }
    comp.push_back(S.top());
    S.pop();
    ans.push_back(comp);
    low[u] = V + 1;
  }
}

vector<vector<int>> scc(){
  vector<int> low(V), label(V);
  int time = 0;
  vector<vector<int>> ans;
  stack<int> S;
  for(int u = 0; u < V; ++u)
    if(!label[u]) scc(u, low, label, time, ans, S);
  return ans;
}
```

6.15. Árbol mínimo de expansión (Kruskal)

```
vector<edge> kruskal(){
  sort(edges.begin(), edges.end());
  vector<edge> MST;
  disjointSet DS(V);
  for(int u = 0; u < V; ++u)
    DS.makeSet(u);
  int i = 0;
  while(i < edges.size() && MST.size() < V - 1){
    edge current = edges[i++];
    int u = current.source, v = current.dest;
    if(DS.findSet(u) != DS.findSet(v)){
      MST.push_back(current);
      DS.unionSet(u, v);
    }
  }
  return MST;
}
```

6.16. Máximo emparejamiento bipartito

```
bool tryKuhn(int u, vector<bool> & used, vector<int> & left,
↪ vector<int> & right){
  if(used[u]) return false;
  used[u] = true;
  for(edge & current : adjList[u]){
    int v = current.dest;
    if(right[v] == -1 || tryKuhn(right[v], used, left,
    ↪ right)){
      right[v] = u;
      left[u] = v;
      return true;
    }
  }
  return false;
}

bool augmentingPath(int u, vector<bool> & used, vector<int> &
↪ left, vector<int> & right){
```

```
    used[u] = true;
    for(edge & current : adjList[u]){
      int v = current.dest;
      if(right[v] == -1){
        right[v] = u;
        left[u] = v;
        return true;
      }
    }
    for(edge & current : adjList[u]){
      int v = current.dest;
      if(!used[right[v]] && augmentingPath(right[v], used,
      ↪  left, right)){
        right[v] = u;
        left[u] = v;
        return true;
      }
    }
    return false;
  }

  //vertices from the left side numbered from 0 to l-1
  //vertices from the right side numbered from 0 to r-1
  //graph[u] represents the left side
  //graph[u][v] represents the right side
  //we can use tryKuhn() or augmentingPath()
  vector<pair<int, int>> maxMatching(int l, int r){
    vector<int> left(l, -1), right(r, -1);
    vector<bool> used(l, false);
    for(int u = 0; u < l; ++u){
      tryKuhn(u, used, left, right);
      fill(used.begin(), used.end(), false);
    }
    vector<pair<int, int>> ans;
    for(int u = 0; u < r; ++u){
      if(right[u] != -1){
        ans.push_back({right[u], u});
      }
    }
    return ans;
  }
```

7. Árboles

7.1. Estructura tree

```
struct tree{
  vector<int> parent, level, weight;
  vector<vector<int>> dists, DP;
  int n, root;

  void graph_to_tree(int prev, int u, graph & G){
    for(edge & curr : G.adjList[u]){
      int v = curr.dest;
      int w = curr.cost;
      if(v == prev) continue;
      parent[v] = u;
      weight[v] = w;
      graph_to_tree(u, v, G);
    }
  }

  int dfs(int i){
    if(i == root) return 0;
    if(level[parent[i]] != -1) return level[i] = 1 +
    ↪  level[parent[i]];
    return level[i] = 1 + dfs(parent[i]);
  }

  void buildLevels(){
    for(int i = n - 1; i >= 0; --i){
      if(level[i] == -1){
        level[i] = dfs(i);
      }
    }
  }

  tree(int n, int root){
    this->n = n;
    this->root = root;
    parent.resize(n);
    level.resize(n, -1);
```

```
    weight.resize(n);
    dists.resize(n, vector<int>(20));
    DP.resize(n, vector<int>(20));
    level[root] = 0;
    parent[root] = root;
  }

  tree(graph & G, int root){
    tree(G.V, root);
    graph_to_tree(-1, root, G);
    buildLevels();
  }

  void pre(){
    for(int u = 0; u < n; u++){
      DP[u][0] = parent[u];
      dists[u][0] = weight[u];
    }
    for(int i = 1; (1 << i) <= n; ++i){
      for(int u = 0; u < n; ++u){
        DP[u][i] = DP[DP[u][i - 1]][i - 1];
        dists[u][i] = dists[u][i - 1] + dists[DP[u][i - 1]][i -
        ↪  1];
      }
    }
  }
```

7.2. k-ésimo ancestro

```
int ancestor(int p, int k){
  int h = level[p] - k;
  if(h < 0) return -1;
  int lg;
  for(lg = 1; (1 << lg) <= level[p]; ++lg);
  lg--;
  for(int i = lg; i >= 0; --i){
    if(level[p] - (1 << i) >= h){
      p = DP[p][i];
    }
  }
  return p;
}
```

7.3. LCA

```
int lca(int p, int q){
  if(level[p] < level[q]) swap(p, q);
  int lg;
  for(lg = 1; (1 << lg) <= level[p]; ++lg);
  lg--;
  for(int i = lg; i >= 0; --i){
    if(level[p] - (1 << i) >= level[q]){
      p = DP[p][i];
    }
  }
  if(p == q) return p;

  for(int i = lg; i >= 0; --i){
    if(DP[p][i] != -1 && DP[p][i] != DP[q][i]){
      p = DP[p][i];
      q = DP[q][i];
    }
  }
  return parent[p];
}
```

7.4. Distancia entre dos nodos

```
int dist(int p, int q){
  if(level[p] < level[q]) swap(p, q);
  int lg;
  for(lg = 1; (1 << lg) <= level[p]; ++lg);
  lg--;
  int sum = 0;
  for(int i = lg; i >= 0; --i){
    if(level[p] - (1 << i) >= level[q]){
      sum += dists[p][i];
      p = DP[p][i];
    }
```

```
    }
    if(p == q) return sum;

    for(int i = lg; i >= 0; --i){
      if(DP[p][i] != -1 && DP[p][i] != DP[q][i]){
        sum += dists[p][i] + dists[q][i];
        p = DP[p][i];
        q = DP[q][i];
      }
    }
    sum += dists[p][0] + dists[q][0];
    return sum;
  }
```

8. Flujos

8.1. Estructura flowEdge

```
template<typename T>
struct flowEdge{
  int dest;
  T flow, capacity, cost;
  flowEdge *res;
  flowEdge(){
    this->dest = this->flow = this->capacity = this->cost = 0;
    this->res = NULL;
  }
  flowEdge(int dest, T flow, T capacity, T cost = 0){
    this->dest = dest, this->flow = flow, this->capacity =
    ↪  capacity, this->cost = cost;
    this->res = NULL;
  }
  void addFlow(T flow){
    this->flow += flow;
    this->res->flow -= flow;
  }
};
```

8.2. Estructura flowGraph

```
template<typename T>
struct flowGraph{
  T inf = numeric_limits<T>::max();
  vector<vector<flowEdge<T>*>> adjList;
  vector<int> dist, pos;
  int V;
  flowGraph(int V){
    this->V = V;
    adjList.resize(V);
    dist.resize(V);
    pos.resize(V);
  }
  ~flowGraph(){
    for(int i = 0; i < V; ++i)
```

```
    for(int j = 0; j < adjList[i].size(); ++j)
      delete adjList[i][j];
  }
  void addEdge(int u, int v, T capacity, T cost = 0){
    flowEdge<T> *uv = new flowEdge<T>(v, 0, capacity, cost);
    flowEdge<T> *vu = new flowEdge<T>(u, capacity, capacity,
    ↪  -cost);
    uv->res = vu;
    vu->res = uv;
    adjList[u].push_back(uv);
    adjList[v].push_back(vu);
  }
```

8.3. Algoritmo de Edmonds-Karp $O(VE^2)$

```
  //Maximun Flow using Edmonds-Karp Algorithm O(VE^2)
  T edmondsKarp(int s, int t){
    T maxFlow = 0;
    vector<flowEdge<T>*> parent(V);
    while(true){
      fill(parent.begin(), parent.end(), nullptr);
      queue<int> Q;
      Q.push(s);
      while(!Q.empty() && !parent[t]){
        int u = Q.front(); Q.pop();
        for(flowEdge<T> *v : adjList[u]){
          if(!parent[v->dest] && v->capacity > v->flow){
            parent[v->dest] = v;
            Q.push(v->dest);
          }
        }
      }
      if(!parent[t]) break;
      T f = inf;
      for(int u = t; u != s; u = parent[u]->res->dest)
        f = min(f, parent[u]->capacity - parent[u]->flow);
      for(int u = t; u != s; u = parent[u]->res->dest)
        parent[u]->addFlow(f);
      maxFlow += f;
    }
    return maxFlow;
  }
```

8.4. Algoritmo de Dinic $O(V^2E)$

```
  //Maximun Flow using Dinic Algorithm O(EV^2)
  T blockingFlow(int u, int t, T flow){
    if(u == t) return flow;
    for(int &i = pos[u]; i < adjList[u].size(); ++i){
      flowEdge<T> *v = adjList[u][i];
      if(v->capacity > v->flow && dist[u] + 1 ==
      ↪  dist[v->dest]){
        T fv = blockingFlow(v->dest, t, min(flow, v->capacity -
        ↪  v->flow));
        if(fv > 0){
          v->addFlow(fv);
          return fv;
        }
      }
    }
    return 0;
  }
  T dinic(int s, int t){
    T maxFlow = 0;
    dist[t] = 0;
    while(dist[t] != -1){
      fill(dist.begin(), dist.end(), -1);
      queue<int> Q;
      Q.push(s);
      dist[s] = 0;
      while(!Q.empty()){
        int u = Q.front(); Q.pop();
        for(flowEdge<T> *v : adjList[u]){
          if(dist[v->dest] == -1 && v->flow != v->capacity){
            dist[v->dest] = dist[u] + 1;
            Q.push(v->dest);
          }
        }
      }
      if(dist[t] != -1){
```

```
      T f;
      fill(pos.begin(), pos.end(), 0);
      while(f = blockingFlow(s, t, inf))
        maxFlow += f;
    }
  }
  return maxFlow;
}
```

8.5. Flujo máximo de costo mínimo

```
//Max Flow Min Cost
pair<T, T> maxFlowMinCost(int s, int t){
  vector<bool> inQueue(V);
  vector<T> distance(V), cap(V);
  vector<flowEdge<T>*> parent(V);
  T maxFlow = 0, minCost = 0;
  while(true){
    fill(distance.begin(), distance.end(), inf);
    fill(parent.begin(), parent.end(), nullptr);
    fill(cap.begin(), cap.end(), 0);
    distance[s] = 0;
    cap[s] = inf;
    queue<int> Q;
    Q.push(s);
    while(!Q.empty()){
      int u = Q.front(); Q.pop(); inQueue[u] = 0;
      for(flowEdge<T> *v : adjList[u]){
        if(v->capacity > v->flow && distance[v->dest] >
        ↪  distance[u] + v->cost){
          distance[v->dest] = distance[u] + v->cost;
          parent[v->dest] = v;
          cap[v->dest] = min(cap[u], v->capacity - v->flow);
          if(!inQueue[v->dest]){
            Q.push(v->dest);
            inQueue[v->dest] = true;
          }
        }
      }
    }
    if(!parent[t]) break;
    maxFlow += cap[t];
    minCost += cap[t] * distance[t];
    for(int u = t; u != s; u = parent[u]->res->dest)
      parent[u]->addFlow(cap[t]);
  }
  return {maxFlow, minCost};
}
```

9. Estructuras de datos

9.1. Segment Tree

```
template<typename T>
struct SegmentTree{
  int N;
  vector<T> ST;

  SegmentTree(int N){
    this->N = N;
    ST.assign(N << 1, 0);
  }

  void build(vector<T> & arr){
    for(int i = 0; i < N; ++i)
      ST[N + i] = arr[i];
    for(int i = N - 1; i > 0; --i)
      ST[i] = ST[i << 1] + ST[i << 1 | 1];
  }

  //single element update in pos
  void update(int pos, T value){
    ST[pos += N] = value;
    while(pos >>= 1)
      ST[pos] = ST[pos << 1] + ST[pos << 1 | 1];
  }

  //single element update in [l, r]
  void update(int l, int r, T value){
    l += N, r += N;
    for(int i = l; i <= r; ++i)
      ST[i] = value;
    l >>= 1, r >>= 1;
    while(l >= 1){
      for(int i = r; i >= l; --i)
        ST[i] = ST[i << 1] + ST[i << 1 | 1];
      l >>= 1, r >>= 1;
    }
  }

  //range query, [l, r]
  T query(int l, int r){
    T res = 0;
    for(l += N, r += N; l <= r; l >>= 1, r >>= 1) {
      if(l & 1) res += ST[l++];
      if(!(r & 1)) res += ST[r--];
    }
    return res;
  }
};
```

9.2. Fenwick Tree

```
template<typename T>
struct FenwickTree{
  int N;
  vector<T> bit;

  FenwickTree(int N){
    this->N = N;
    bit.assign(N, 0);
  }

  void build(vector<T> & arr){
    for(int i = 0; i < arr.size(); ++i){
      update(i, arr[i]);
    }
  }

  //single element increment
  void update(int pos, T value){
    while(pos < N){
      bit[pos] += value;
      pos |= pos + 1;
    }
  }

  //range query, [0, r]
  T query(int r){
    T res = 0;
```

```
    while(r >= 0){
      res += bit[r];
      r = (r & (r + 1)) - 1;
    }
    return res;
  }

  //range query, [l, r]
  T query(int l, int r){
    return query(r) - query(l - 1);
  }
};
```