Team reference

Bagel Flavored Debugging
New York University

General

Template

```
//#include <iostream, cstdio, cstdlib, cmath, ctime, cassert>
//#include <cstring, string, algorithm, sstream, complex>
//#include <stack, queue, vector, set, map>
using namespace std;
typedef long long ll;
typedef vector<int> VI;
typedef pair<int,int> PII;
#define REP(i,s,t) for(int i=(s);i<(t);i++)
#define FILL(x,v) memset(x,v,sizeof(x))
```

Checker script

```
# usage: ./check A [testcasenr]

g++ $1.cpp -o $1 -O2 -Wall -Wextra -std=gnu++0x || exit

for i in $1.in${2-*}; do
    echo "### $i ###"
    /usr/bin/time -f "%U s, %M kb" ./$1 < $i > ${i/in/out}
    diff -q ${i/in/ok} ${i/in/out} || diff -y ${i/in/ok} ${i/in/out}
done
```

Common primes

10.007, 100.000.007, 1.000.000.007, 99.991, 9.999.991

Maximum values

int: at least up to $2,1 \cdot 10^9$
long long: at least up to $9,2 \cdot 10^{18}$

Contents

1 Combinatorics

1.1 Pólya enumeration theorem

Let the group G act on the set X. The number of G-orbits of colorings of X with k colors (i.e., the number of colorings of X with n colors where colorings are identified if one can be obtained from the other by permuting the elements of X via an element of the group G) is

$$|\{1,\dots,k\}^X/G| = \frac{1}{|G|}\sum_{g\in G} k^{c(g)}$$

if $c(g)$ is the number of (possibly one-element) cycles in the permutation of X given by $g \in G$. For example, the number of colorings of a necklace of n beads with k colors is

$$c(n,k) := \frac{1}{n}\sum_{i=0}^{n-1} k^{\gcd(i,n)} = \frac{1}{n}\sum_{d|n}\varphi(n/d)k^d$$

if we consider colorings equal if one can be obtained from the other by a 2D rotation.

1.2 Twelvefold way

Wanted: Number of maps $f\colon N \to X$ from an n-element set into an x-element set with specific properties.

	arbitrary f	injective f	surjective f
Order matters	x^n	$x^{\underline{n}}$	$x! \cdot \left\{{n \atop x}\right\}$
Order of N irrelevant	$\binom{x+n-1}{n}$	$\binom{x}{n}$	$\binom{n-1}{n-x}$
Order of X irrelevant	$\sum_{k=0}^{x}\left\{{n \atop k}\right\}$	$[n \le x]$	$\left\{{n \atop x}\right\}$
Orders of N and X irrelevant	$p_x(n+x)$	$[n \le x]$	$p_x(n)$

Notation:

- The falling factorial $x^{\underline{n}} = x(x-1)(x-2)\cdots(x-n+1)$.
- The factorial $n! = n^{\underline{n}} = n(n-1)(n-2)\cdots 1$.
- The Stirling number of the second kind $\left\{{n \atop k}\right\}$, which is the number of partitions of an n-element set into k non-empty sets.
- The binomial coefficient $\binom{n}{k} = \frac{n^{\underline{k}}}{k!}$.
- The indicator function $[p] = 0$ if p is false, $[p] = 1$ if p is true.
- The number of partitions $p_k(n)$ of a number n into k summands.

1.3 Table with common sequences

B_n is the number of partitions of a set with n elements, i.e. the number of equivalence relations on the set $\{1,\dots,n\}$.
p_n is the number of partitions of n, i.e. the number of representations of n as the sum of positive integers (where the order is irrelevant).
$C_n = \frac{1}{n+1}\binom{2n}{n}$ is the number of binary trees, in which every node has 0 or 2 children. Furthermore, C_n is the number of words of length $2n$ consisting of n characters A and n characters B, such that before each point in the string, there are at most as many Bs as As. These numbers fulfill the recursion relations $C_0 = 1$ and $C_{n+1} = \sum_{i=0}^{n} C_i C_{n-i}$.

n	2^n	3^n	$n!$	B_n	p_n	$\binom{n}{\lfloor n/2 \rfloor}$
0	1	1	1	1	1	1
1	2	3	1	1	1	1
2	4	9	2	2	2	2
3	8	27	6	5	3	3
4	16	81	24	15	5	6
5	32	$\leq 2,5 \cdot 10^{2}$	$\leq 1,2 \cdot 10^{2}$	52	7	10
6	64	$\leq 7,3 \cdot 10^{2}$	$\leq 7,2 \cdot 10^{2}$	$\leq 2,1 \cdot 10^{2}$	11	20
7	$\leq 1,3 \cdot 10^{2}$	$\leq 2,2 \cdot 10^{3}$	$\leq 5,1 \cdot 10^{3}$	$\leq 8,8 \cdot 10^{2}$	15	35
8	$\leq 2,6 \cdot 10^{2}$	$\leq 6,6 \cdot 10^{3}$	$\leq 4,1 \cdot 10^{4}$	$\leq 4,2 \cdot 10^{3}$	22	70
9	$\leq 5,2 \cdot 10^{2}$	$\leq 2,0 \cdot 10^{4}$	$\leq 3,7 \cdot 10^{5}$	$\leq 2,2 \cdot 10^{4}$	30	$\leq 1,3 \cdot 10^{2}$
10	$\leq 1,1 \cdot 10^{3}$	$\leq 6,0 \cdot 10^{4}$	$\leq 3,7 \cdot 10^{6}$	$\leq 1,2 \cdot 10^{5}$	42	$\leq 2,6 \cdot 10^{2}$
11	$\leq 2,1 \cdot 10^{3}$	$\leq 1,8 \cdot 10^{5}$	$\leq 4,0 \cdot 10^{7}$	$\leq 6,8 \cdot 10^{5}$	56	$\leq 4,7 \cdot 10^{2}$
12	$\leq 4,1 \cdot 10^{3}$	$\leq 5,4 \cdot 10^{5}$	$\leq 4,8 \cdot 10^{8}$	$\leq 4,3 \cdot 10^{6}$	77	$\leq 9,3 \cdot 10^{2}$
13	$\leq 8,2 \cdot 10^{3}$	$\leq 1,6 \cdot 10^{6}$	$\leq 6,3 \cdot 10^{9}$	$\leq 2,8 \cdot 10^{7}$	$\leq 1,1 \cdot 10^{2}$	$\leq 1,8 \cdot 10^{3}$
14	$\leq 1,7 \cdot 10^{4}$	$\leq 4,8 \cdot 10^{6}$	$\leq 8,8 \cdot 10^{10}$	$\leq 2,0 \cdot 10^{8}$	$\leq 1,4 \cdot 10^{2}$	$\leq 3,5 \cdot 10^{3}$
15	$\leq 3,3 \cdot 10^{4}$	$\leq 1,5 \cdot 10^{7}$	$\leq 1,4 \cdot 10^{12}$	$\leq 1,4 \cdot 10^{9}$	$\leq 1,8 \cdot 10^{2}$	$\leq 6,5 \cdot 10^{3}$
16	$\leq 6,6 \cdot 10^{4}$	$\leq 4,4 \cdot 10^{7}$	$\leq 2,1 \cdot 10^{13}$	$\leq 1,1 \cdot 10^{10}$	$\leq 2,4 \cdot 10^{2}$	$\leq 1,3 \cdot 10^{4}$
17	$\leq 1,4 \cdot 10^{5}$	$\leq 1,3 \cdot 10^{8}$	$\leq 3,6 \cdot 10^{14}$	$\leq 8,3 \cdot 10^{10}$	$\leq 3,0 \cdot 10^{2}$	$\leq 2,5 \cdot 10^{4}$
18	$\leq 2,7 \cdot 10^{5}$	$\leq 3,9 \cdot 10^{8}$	$\leq 6,5 \cdot 10^{15}$	$\leq 6,9 \cdot 10^{11}$	$\leq 3,9 \cdot 10^{2}$	$\leq 4,9 \cdot 10^{4}$
19	$\leq 5,3 \cdot 10^{5}$	$\leq 1,2 \cdot 10^{9}$	$\leq 1,3 \cdot 10^{17}$	$\leq 5,9 \cdot 10^{12}$	$\leq 4,9 \cdot 10^{2}$	$\leq 9,3 \cdot 10^{4}$
20	$\leq 1,1 \cdot 10^{6}$	$\leq 3,5 \cdot 10^{9}$	$\leq 2,5 \cdot 10^{18}$	$\leq 5,2 \cdot 10^{13}$	$\leq 6,3 \cdot 10^{2}$	$\leq 1,9 \cdot 10^{5}$
21	$\leq 2,1 \cdot 10^{6}$	$\leq 1,1 \cdot 10^{10}$	$\leq 5,2 \cdot 10^{19}$	$\leq 4,8 \cdot 10^{14}$	$\leq 8,0 \cdot 10^{2}$	$\leq 3,6 \cdot 10^{5}$
22	$\leq 4,2 \cdot 10^{6}$	$\leq 3,2 \cdot 10^{10}$	$\leq 1,2 \cdot 10^{21}$	$\leq 4,6 \cdot 10^{15}$	$\leq 1,1 \cdot 10^{3}$	$\leq 7,1 \cdot 10^{5}$
23	$\leq 8,4 \cdot 10^{6}$	$\leq 9,5 \cdot 10^{10}$	$\leq 2,6 \cdot 10^{22}$	$\leq 4,5 \cdot 10^{16}$	$\leq 1,3 \cdot 10^{3}$	$\leq 1,4 \cdot 10^{6}$
24	$\leq 1,7 \cdot 10^{7}$	$\leq 2,9 \cdot 10^{11}$	$\leq 6,3 \cdot 10^{23}$	$\leq 4,5 \cdot 10^{17}$	$\leq 1,6 \cdot 10^{3}$	$\leq 2,8 \cdot 10^{6}$
25	$\leq 3,4 \cdot 10^{7}$	$\leq 8,5 \cdot 10^{11}$	$\leq 1,6 \cdot 10^{25}$	$\leq 4,7 \cdot 10^{18}$	$\leq 2,0 \cdot 10^{3}$	$\leq 5,3 \cdot 10^{6}$
26	$\leq 6,8 \cdot 10^{7}$	$\leq 2,6 \cdot 10^{12}$	$\leq 4,1 \cdot 10^{26}$	$\leq 5,0 \cdot 10^{19}$	$\leq 2,5 \cdot 10^{3}$	$\leq 1,1 \cdot 10^{7}$
27	$\leq 1,4 \cdot 10^{8}$	$\leq 7,7 \cdot 10^{12}$	$\leq 1,1 \cdot 10^{28}$	$\leq 5,5 \cdot 10^{20}$	$\leq 3,1 \cdot 10^{3}$	$\leq 2,1 \cdot 10^{7}$
28	$\leq 2,7 \cdot 10^{8}$	$\leq 2,3 \cdot 10^{13}$	$\leq 3,1 \cdot 10^{29}$	$\leq 6,2 \cdot 10^{21}$	$\leq 3,8 \cdot 10^{3}$	$\leq 4,1 \cdot 10^{7}$
29	$\leq 5,4 \cdot 10^{8}$	$\leq 6,9 \cdot 10^{13}$	$\leq 8,9 \cdot 10^{30}$	$\leq 7,2 \cdot 10^{22}$	$\leq 4,6 \cdot 10^{3}$	$\leq 7,8 \cdot 10^{7}$
30	$\leq 1,1 \cdot 10^{9}$	$\leq 2,1 \cdot 10^{14}$	$\leq 2,7 \cdot 10^{32}$	$\leq 8,5 \cdot 10^{23}$	$\leq 5,7 \cdot 10^{3}$	$\leq 1,6 \cdot 10^{8}$
31	$\leq 2,2 \cdot 10^{9}$	$\leq 6,2 \cdot 10^{14}$	$\leq 8,3 \cdot 10^{33}$	$\leq 1,1 \cdot 10^{25}$	$\leq 6,9 \cdot 10^{3}$	$\leq 3,1 \cdot 10^{8}$
32	$\leq 4,3 \cdot 10^{9}$	$\leq 1,9 \cdot 10^{15}$	$\leq 2,7 \cdot 10^{35}$	$\leq 1,3 \cdot 10^{26}$	$\leq 8,4 \cdot 10^{3}$	$\leq 6,1 \cdot 10^{8}$
33	$\leq 8,6 \cdot 10^{9}$	$\leq 5,6 \cdot 10^{15}$	$\leq 8,7 \cdot 10^{36}$	$\leq 1,7 \cdot 10^{27}$	$\leq 1,1 \cdot 10^{4}$	$\leq 1,2 \cdot 10^{9}$
34	$\leq 1,8 \cdot 10^{10}$	$\leq 1,7 \cdot 10^{16}$	$\leq 3,0 \cdot 10^{38}$	$\leq 2,2 \cdot 10^{28}$	$\leq 1,3 \cdot 10^{4}$	$\leq 2,4 \cdot 10^{9}$
35	$\leq 3,5 \cdot 10^{10}$	$\leq 5,1 \cdot 10^{16}$	$\leq 1,1 \cdot 10^{40}$	$\leq 2,9 \cdot 10^{29}$	$\leq 1,5 \cdot 10^{4}$	$\leq 4,6 \cdot 10^{9}$

1.4 Number k-element subsets

```
// Returns the a-th way (counting from 0) to choose a k-element subset from an n-element set.
// The subset will be encoded using a bitmask.
ll integerToSubset(int n, int k, ll a) {
    ll bits = 0;
    for(int m = n - 1; m >= 0; --m)
        if(a >= binom[m][k]) {
            a -= binom[m][k];
            bits |= 1ll << m;
            --k;
        }
    return bits;
}

// The inverse function to integerToSubset. Returns the number encoding a subset.
ll subsetToInteger(int n, ll bits) {
    ll a = 0, j = 0;
    REP(m,0,n)
        if(bits & (1ll << m)) {
            ++j;
            a += binom[m][j];
        }
    return a;
}
```

1.5 Number permutations

```
// Converts a number s ∈ {0,...,n!-1} to a permutation perm of the numbers {0,...,n-1}.
void integerToPermutation(ll s, int *perm, int n) {
    for(int i = n - 1; i >= 0; --i) {
        perm[i] = s % (n - i);
        s /= (n - i);
        REP(j,i + 1,n)
            if(perm[j] >= perm[i])
                ++perm[j];
    }
}

// The inverse function to integerToPermutation. Returns the number encoding a permutation.
ll permutationToInteger(const int *perm, int n) {
    ll s = 0;
    REP(i,0,n) {
        s = s * (n - i) + perm[i];
        REP(j,0,i)
            if(perm[j] < perm[i])
                --s;
    }
    return s;
}
```

2 Number theory

2.1 Extended euclidean algorithm/invert modulo n

Running time: $\mathcal{O}(\log(\min(a,b)))$
Warning: Only call this function for *positive* numbers a and b!
Afterwards, $ad + be = c$ and $c = \gcd(a,b)$.

```
void exteuclid(ll a, ll b, ll &c, ll &d, ll &e) {
    if (b == 0) {
        c = a;
        d = 1;
        e = 0;
    } else {
        exteuclid(b, a%b, c, e, d);
```

```
        e -= d*(a/b);
    }
}
```

Returns $a^{-1} \mod m$ for **relatively prime positive** numbers a, m.

```
ll invmod(ll a, ll m) {
    ll c, d, e;
    exteuclid(a, m, c, d, e);
    return mod(d,m);
}
```

2.2 Slow multiplication

Returns $ab \mod m$

```
ll multmod(ll a, ll b, ll m) {
    ll e = a, r = 0;
    for (int i = 0; (1ll<<i) <= b; i++) { // "one el el"
        if (b&(1ll<<i))
            r = (r+e)%m;
        e = 2*e%m;
    }
    return r;
}
```

2.3 Fast exponentiation

Returns $a^b \mod m$ (Warning: beware of overflows if m is large).

```
ll powermod(ll a, ll b, ll m) {
    ll e = a, r = 1;
    for (int i = 0; (1ll<<i) <= b; i++) { // "one el el"
        if (b&(1ll<<i))
            r = r*e%m; // use multmod if necessary
        e = e*e%m; // use multmod if necessary
    }
    return r;
}
```

You can easily calculate the inverse of a modulo a **prime** p: `powermod(a,p-2,p)`

2.4 Precompute inverses modulo primes

Works only for prime numbers!

```
int inv[...];
void calcinvs(int P) {
    inv[1] = 1;
    REP(k,2,P)
        inv[k] = (-(ll)(P/k)*inv[P%k]%P+P)%P;
}
```

2.5 Find primes, precompute smallest prime divisors and Euler's totient function

Running time: $\mathcal{O}(n \log n)$

```
int dvd[...]; // The smallest prime dividing this number (-1 for 0 and 1); the number n is prime iff dvd[n] == n
VI prim; // List of the prime numbers
int phi[...]; // φ(k) is the number of integers 1 ≤ a ≤ k relatively prime to k for k ≥ 1

void findprimes(int n) { // Precomputes smallest divisors and φ(k) for 0 ≤ k ≤ n
    REP(i,2,n+1)
        dvd[i] = i;
    dvd[0] = dvd[1] = -1;
    phi[1] = 1;
    prim.clear();
    for (ll k = 2; k <= n; k++) {
        if (dvd[k] == k) {
            prim.push_back(k);
            for (ll t = k*k; t <= n; t += k)
                if (dvd[t] == t)
                    dvd[t] = k;
        }
        // compute φ(k)
        int r = k;
        while(r%dvd[k] == 0)
            r /= dvd[k];
        phi[k] = phi[r]*(k/r-k/r/dvd[k]);
    }
}
```

2.6 Finding a Primitive Root

```
ll fac[1000];
ll primroot(ll n) { // returns the smallest primitive root a ≥ 1 modulo n if n is prime
    ll x = n-1, ph = n-1; // if n has a primitive root (i.e., n = 2,4,p^k,2p^k for some odd prime p) but is not necessarily prime, replace n-1 by phi[n]
    int L = 0;
    while(x>1){
        int p = dvd[x];
        fac[L++] = p; // get all factors
        while(x%p==0) x/=p;
    }
    for(ll g=1; g<=n; g++){
        bool ok = 1;
        for(int i=0; i<L && ok; i++)
            if(powermod(g, ph/fac[i], n)==1) ok = 0;
        if(ok) return g;
    }
    return -1;
}
```

2.7 Miller-Rabin Primality Test

Returns whether n is prime in time $\mathcal{O}(\log n)$ (Warning: beware of overflows if n is large).

```
const ll testa[] = {2,7,61}; // would work for n < 4 759 123 141; add random numbers for larger n
bool isprime(ll n) {
    if(n < 2)
        return false;
    ll d = n-1;
    int j = 0;
    while(d%2 == 0) {
        d /= 2;
        j++;
    }
    for (ll a : testa) {
        if (n == a)
            return true;
        ll e = powermod(a,d,n); // use multmod inside if necessary
        bool ok = (e == 1);
        REP(r,0,j) {
            if (e == n-1)
                ok = true;
            e = multmod(e,e,n); // use e*e%n if ok
        }
        if (!ok)
            return false;
    }
    return true;
}
```

2.8 Pollard's rho algorithm

```
ll pollard_rho(ll n){
    int i = 1, k = 2;
    ll x = ((ll)rand()*rand())%n, y = x, d;
    do {
        i++;
        d = gcd(n+y-x,n);
        if(d>1 && d<n) return d;
        if(i==k){ y = x; k <<= 1; }
        x = (multmod(x,x,n)+n-1)%n;
    } while(x != y);
    return n;
}

// smallestFactor(n) returns the smallest prime dividing the integer n > 1
ll smallestFactor(ll n){
    if(n == 1 || isprime(n))
        return n;
    ll f=n;
    while(f==n) f = pollard_rho(n);
    ll f1 = smallestFactor(f), f2 = smallestFactor(n/f);
    if (f1 == 1) return f2;
    if (f2 == 1) return f1;
    return min(f1, f2);
}
```

3 Big integers

Obviously, you have to type in only parts of the following code.

```
typedef vector<ll> VLL;
const int ZIFSI=1E9;
struct gr {
    VLL z; // The number is sum_{i=0}^{z.size()-1} z_i ZIFSI^i
    gr() {
    }
    gr(ll a) {
        z.push_back(a);
        can();
    }
    // Canonicalizes the number, i.e.:
    // a) -ZIFSI < z_i < ZIFSI for all i
    // b) the numbers z_i are all >= 0 or all <= 0
    // c) z.back() != 0 or z.size() = 0
    gr& can() {
        ll u = 0;
        REP(i,0,(int)z.size()) {
            z[i] += u;
            u = z[i]/ZIFSI;
            z[i] %= ZIFSI;
        }
        z.push_back(u);
        while(z.size() && z.back() == 0)
            z.pop_back();
        if (z.size()) { // only necessary if numbers can be negative
            int s = z.back() > 0 ? 1 : -1;
            REP(i,0,(int)z.size()) {
                if (z[i]*s < 0) {
                    z[i] += s*ZIFSI;
                    z[i+1] -= s;
                }
            }
            while(z.size() && z.back() == 0)
                z.pop_back();
        }
        return *this;
    }
    // Returns 0 if i >=z.size(); to be safe, better use this than .z[...]
    ll operator[](int i) const {
        return i < (int)z.size() ? z[i] : 0;
    }
    gr& operator +=(const gr &a) {
        if (z.size() < a.z.size())
            z.resize(a.z.size());
        REP(i,0,(int)a.z.size())
            z[i] += a.z[i];
        return can();
    }
    gr operator -() const {
        gr a;
        for (ll t : z)
            a.z.push_back(-t);
```

```
        return a.can();
    }
    gr& operator -=(const gr &a) {
        return *this += -a;
    }

    gr shift(int k) const { // just for Karatsuba multiplication
        gr a;
        a.z.assign(k, 0);
        a.z.insert(a.z.end(), z.begin(), z.end());
        return a;
    }
    gr slice(int a, int b) const { // just for Karatsuba multiplication
        gr r;
        r.z.insert(r.z.end(), z.begin() + a, z.begin() + b);
        return r;
    }
};
gr operator+(const gr &a, const gr &b) {
    return gr(a)+=b;
}
gr operator-(const gr &a, const gr &b) {
    return gr(a)-=b;
}
// Running time: O(nm) if the numbers have n and m digits
// (keep in mind that this "is" just a ninth of the number of decimal digits)
gr operator*(const gr &a, const gr &b) {
    if ((int)a.z.size() < (int)b.z.size())
        return b*a;
    gr erg;
    REP(i,0,(int)b.z.size()) {
        gr v;
        v.z.assign(i,0);
        REP(j,0,(int)a.z.size())
            v.z.push_back(a.z[j]*b.z[i]);
        erg += v.can();
    }
    return erg;
}
gr& operator *=(gr &a, const gr &b) {
    return a = a*b;
}
// return values: -1 <=> a < b;  0 <=> a = b;  +1 <=> a > b
int cmp(const gr &a, const gr &b) {
    for (int i = max(a.z.size(),b.z.size()); i >= 0; i--) {
        if (a[i] < b[i])
            return -1;
        if (a[i] > b[i])
            return 1;
    }
    return 0;
}
bool operator==(const gr &a, const gr &b) {
    return cmp(a,b) == 0;
}
bool operator!=(const gr &a, const gr &b) {
    return cmp(a,b) != 0;
}
bool operator<=(const gr &a, const gr &b) {
    return cmp(a,b) <= 0;
}
bool operator<(const gr &a, const gr &b) {
    return cmp(a,b) < 0;
}
bool operator>=(const gr &a, const gr &b) {
    return cmp(a,b) >= 0;
}
bool operator>(const gr &a, const gr &b) {
    return cmp(a,b) > 0;
}
void print(const gr &a) { // Print the number
    if (a.z.size() == 0)
        printf("0");
    else {
        printf("%lld", a.z.back()); // percent el el de
        for (int i = (int)a.z.size()-2; i >= 0; i--)
            printf("%09lld", abs(a.z[i])); // percent zero nine el el de
    }
}
char zeile[...]; // Maximum length of an input string plus 20
gr read() { // Read a number
    fill_n(zeile, 10, '0');
    scanf("%s", zeile+10);
    gr erg;
    bool pos = true;
    for (int i = strlen(zeile)-1; i >= 10; i -= 9) {
        int a = 0;
        REP(k,i-8,i+1)
            if (zeile[k] == '-')
                pos = false;
            else
                a = a*10 + zeile[k]-'0';
        erg.z.push_back(a);
    }
    return pos ? erg.can() : -erg;
}
gr karatsuba(const gr &a, const gr &b) { // Multiplies two numbers using Karatsuba
    int h = min(a.z.size(), b.z.size()) / 2;
    if(h <= 5) return a * b;
    gr a1 = a.slice(0, h), a2 = a.slice(h, a.z.size());
    gr b1 = b.slice(0, h), b2 = b.slice(h, b.z.size());
    gr u = karatsuba(a1, b1), v = karatsuba(a2, b2);
    gr w = karatsuba((a1 + a2), (b1 + b2));
    return u + (w - u - v).shift(h) + v.shift(h * 2);
}
```

4 Graph theory

In the following, `int N` is usually the number of nodes, `int M` the number of edges, `VI adj[i]` is a list of the nodes j such that there is an edge from i to j or `vector<PII> adj[i]` is a list of the pairs

(j,l) such that there is an edge from i to j of length l. Multiedges are specified in the canonical way.

4.1 Theorems

Hall's marriage theorem:
Bipartite graphs $G = A \cup B$ satisfy:

$$G \text{ has a matching of size } |A| \Leftrightarrow \forall X \subseteq A\colon |N_G(X)| \geq |X|$$

Tutte theorem:
G has a perfect matching if and only if

$$\forall S \subseteq V\colon \text{Number of components of odd size of } G[V \setminus S] \leq \#S$$

König's theorem:
In a bipartite graph, the size of a maximum matching equals the size of a minimum vertex cover.

Dilworth's theorem:
The maximum size of an antichain of $(P, \leq)$ equals the smallest number of chains disjointly covering P.
Chain: Subset $C \subseteq (P, \leq)$ such that $a \leq b$ or $b \leq a$ holds for all $a, b \in C$
Antichain: Subset $C \subseteq (P, \leq)$ such that neither $a \leq b$ nor $b \leq a$ holds for any $a, b \in C$

4.2 Maximum flow

```
const int INF = 1E9; // ∞: be careful to make this big enough!!!
int S; // source
int T; // sink
int FN; // number of nodes
int FM; // number of edges (initialize this to 0)
// ra[a]: edges connected to a (NO MATTER WHICH WAY!!!); clear this in the beginning
VI ra[...];
int kend[...], cap[...]; // size: TWICE the number of edges
// Adds an edge from a to b with capacity c and returns the number of the new edge
int addedge(int a, int b, int c) {
    int i = 2*FM;
    kend[i] = b;
    cap[i] = c;
    ra[a].push_back(i);
    kend[i+1] = a;
    cap[i+1] = 0;
    ra[b].push_back(i+1);
    FM++;
    return i;
}
```

After `solve()`, the flow through edge number `e` (this number is returned from `addedge`) is `cap[e^1]`.

4.2.1 Edmonds-Karp-Algorithm

Running time: $\mathcal{O}(\min(VE^2, FE))$ with

$$F = \begin{cases} \text{maximum flow}, & \text{all capacities are integers} \\ \infty, & \text{else} \end{cases}$$

```
bool fou[...];
PII pre[...];
// Returns the maximum flow from the source to the sink
ll solve() { // reinitialize costs if rerun
    ll totflow = 0;
    while(true) {
        memset(fou, 0, sizeof(fou));
        queue<int> qu;
        qu.push(S);
        fou[S] = true;
        while(!qu.empty()) {
            int i = qu.front();
            qu.pop();
            if (i == T)
                break;
            for (int e : ra[i]) {
                int k = kend[e];
                if (cap[e] > 0 && !fou[k]) {
                    qu.push(k);
                    pre[k] = PII(i,e);
                    fou[k] = true;
                }
            }
        }
        if (!fou[T])
            break;
        int mk = INF;
        for (int i = T; i != S; i = pre[i].first)
            mk = min(mk, cap[pre[i].second]);
        totflow += mk;
        for (int i = T; i != S; i = pre[i].first) {
            cap[pre[i].second] -= mk;
            cap[pre[i].second^1] += mk;
        }
    }
    return totflow;
}
```

4.2.2 Dinic's algorith with gap heuristic

Running time: $\mathcal{O}(\min(V^2E, FE))$ with F as above

```
int dst[...], gap[...];
void bfs(){
    fill_n(dst, FN, FN);
    dst[T] = 0; // bfs from T
    FILL(gap, 0);
    gap[0] = FN; // FN nodes with gap 0
    queue<int> que; que.push(T);
    while(!que.empty()){
        int x = que.front(); que.pop();
        for (int t : ra[x]){
            if(t&1){ // back edge
                int y = kend[t];
                if(dst[y]==FN){
                    dst[y] = dst[x]+1;
```

```
                    gap[dst[y]]++;
                    que.push(y);
                }
            }
        }
    }
}
int dfs(int x, int inflow){
    if(x==T) return inflow;
    int outflow = inflow, minh = FN-1;
    for (int t : ra[x]){
        int y = kend[t];
        if(cap[t] > 0){
            if(dst[x]==dst[y]+1){
                int delta = dfs(y, min(outflow, cap[t]));
                cap[t] -= delta;
                cap[t^1] += delta;
                outflow -= delta;
                if(dst[S]==FN) return inflow-outflow; // no more flow, cannot advance
                if(outflow==0) break;
            }
            minh = min(minh, dst[y]);
        }
    }
    if(inflow==outflow){ // no exit flow possible, relabel
        gap[dst[x]]--;
        if(gap[dst[x]]==0) dst[S] = FN; // exit immediately
        dst[x] = minh + 1;
        gap[dst[x]]++;
    }
    return inflow-outflow;
}
// Returns the maximum flow from the source to the sink
ll solve(){ // reinitialize costs if rerun
    ll maxflow = 0;
    bfs();
    while(dst[S]<FN) maxflow += dfs(S, INF);
    return maxflow;
}
```

4.3 Minimum cut

Compute the maximum flow. If you used the first code, put the vertices i with `fou[i]` in one set and the other vertices in the other set.
Otherwise, start a depth-first search from the source and only use edges having spare capacity (through which you could send more flow). The visited nodes are one set, the non-visited ones the other set.

4.4 Bipartite matching in bipartite graphs

Let the nodes in the both node sets M_1, M_2 be numbered $0, \dots, N_1 - 1$ and $0, \dots, N_2 - 1$, respectively.

4.4.1 Slow

Running time: $\mathcal{O}(VE)$

```
int N1, N2; // Number of nodes in M1 and M2
VI ad1[...]; // ad1[a]: Nodes in M2 that are connected to a ∈ M1
int ni1[...]; // The node in M2 that is matched to a ∈ M1; -1 if unmatched
int ni2[...]; // The node in M1 that is matched to a ∈ M2; -1 if unmatched
bool vis[...];
bool visit(int i) {
    if (vis[i])
        return false;
    vis[i] = true;
    for (int k : ad1[i]) {
        if (k != ni1[i] && (ni2[k] == -1 || visit(ni2[k]))) {
            ni1[i] = k;
            ni2[k] = i;
            return true;
        }
    }
    return false;
}
// Returns the size of the maximum bipartite matching / minimum vertex cover
int solve() {
    fill_n(ni1, N1, -1);
    fill_n(ni2, N2, -1);
    int numpairs = 0, pv;
    do {
        pv = numpairs;
        fill_n(vis, N1, false);
        REP(i,0,N1)
            if (ni1[i] == -1)
                numpairs += visit(i);
    } while(numpairs != pv);
    return numpairs;
}
```

4.4.2 Fast

Running time: $\mathcal{O}(E\sqrt{V})$

```
const int INF = 1E9;
int N1, N2; // Number of nodes in M1 and M2
VI ad1[...]; // ad1[a]: Nodes in M2 that are connected to a ∈ M1
int ni1[...]; // The node in M2 that is matched to a ∈ M1; -1 if unmatched
int ni2[...]; // The node in M1 that is matched to a ∈ M2; -1 if unmatched
int dst[...];

bool bfs(){
    queue<int> que;
    REP(i,0,N1){
        if(ni1[i]==-1) { dst[i]=0; que.push(i); }
        else dst[i] = INF;
    }
    int maxdst = INF;
    while(!que.empty()){
        int x = que.front(); que.pop();
```

```
        if(dst[x]==maxdst) continue;
        for (int y : ad1[x]) {
            if(ni2[y]==-1){
                maxdst = dst[x];
            }else if(dst[ni2[y]]==INF){
                dst[ni2[y]] = dst[x]+1;
                que.push(ni2[y]);
            }
        }
    }
    return maxdst < INF;
}
bool dfs(int x){
    for (int y : ad1[x]) {
        if(ni2[y]==-1 || (dst[ni2[y]]==dst[x]+1 && dfs(ni2[y]))){
            ni1[x] = y;
            ni2[y] = x;
            return true;
        }
    }
    dst[x] = INF;
    return false;
}
int solve(){
    REP(i,0,N1) ni1[i] = -1;
    REP(i,0,N2) ni2[i] = -1;
    int ans = 0;
    while(bfs()){
        REP(i,0,N1) if(ni1[i]==-1) ans += dfs(i);
    }
    return ans;
}
```

4.5 Minimum vertex cover

After computing the maximum bipartite matching, a minimum vertex cover is given by the set consisting of

- the nodes i from the first set satisfying `!vis[i]` and
- the nodes k from the second set for which there is a node i from the first set satisfying `vis[i] && ni1[i] == k`.

If you used the fast bipartite matching algorithm, substitute `vis[i]` by `dst[i] < INF`.

4.6 Minimum number of paths covering a directed acyclic graph

From the graph (V, E) construct a bipartite graph (V', E') with $V' = V \times \{0, 1\}$ and $E' = \{((a, 0),(b, 1)) \mid (a, b) \in E\}$. The minimum number of paths is $|V| - |M|$ if M is the set of matched edges. An edge $((a, 0),(b, 1))$ is contained in M (= matched) if and only if (a, b) is contained in one of those paths (for a fixed set of paths).

4.7 Minimum cost flow

If you want to find the minimum cost for sending a certain fixed flow f through the network, add a super sink and connect the original sink to it by an edge of capacity f and cost 0. Check that the flow returned by `solve()` equals the expected flow f.

```
const int INF = 1E9; // ∞: be careful to make this big enough but not too big!!!
int S; // source
int T; // sink
int FN; // number of nodes
int FM; // number of edges (initialize this to 0)
// ra[a]: edges connected to a (NO MATTER WHICH WAY!!!); clear this in the beginning
VI ra[...];
int kend[...], cap[...], cost[...]; // size: TWICE the number of edges
// Adds an edge from a to b with capacity c and cost d and returns the number of the new edge
int addedge(int a, int b, int c, int d) {
    int i = 2*FM;
    kend[i] = b;
    cap[i] = c;
    cost[i] = d;
    ra[a].push_back(i);
    kend[i+1] = a;
    cap[i+1] = 0;
    cost[i+1] = -d;
    ra[b].push_back(i+1);
    FM++;
    return i;
}
```

After `solve()`, the flow through edge number `e` (this number is returned from `addedge`) is `cap[e^1]`.

4.7.1 Fabian

Running time $\mathcal{O}(VE + FE\log(E))$ if the pursued flow is F. This algorithm only works if there are no negative cycles. If you don't use a `priority_queue` in the Dijkstra algorithm (and try out all nodes in each iteration to find one of minimum distance), you obtain running time $\mathcal{O}(VE + FV^2)$.

```
int pi[...], dist[...];
PII pred[...];
// returns the maximum flow and the minimum cost for this flow
pair<ll,ll> solve() {
    ll totflow = 0, totcost = 0;
    fill_n(pi, FN, INF);
    pi[S] = 0;
    REP(l,0,FN+1)
        REP(a,0,FN)
            for (int e : ra[a])
                if (cap[e] > 0 && pi[kend[e]] > pi[a]+cost[e])
                    pi[kend[e]] = pi[a]+cost[e];
    while(true) {
        fill_n(dist, FN, INF);
        priority_queue<PII> qu;
        qu.push(PII(0,S));
        dist[S] = 0;
        while(!qu.empty()) {
            int i = qu.top().second, d = -qu.top().first;
            qu.pop();
            if (d > dist[i])
                continue;
            for (int e : ra[i]) {
                int k = kend[e];
                int ds = d+cost[e]+pi[i]-pi[k];
                if (cap[e] > 0 && dist[k] > ds) {
```

```
                    dist[k] = ds;
                    qu.push(PII(-ds,k));
                    pred[k] = PII(i,e);
                }
            }
        }
        REP(i,0,FN)
            pi[i] += dist[i];
        if (dist[T] == INF)
            break;
        int mk = INF;
        ll co = 0;
        for (int i = T; i != S; i = pred[i].first) {
            mk = min(mk, cap[pred[i].second]);
            co += cost[pred[i].second];
        }
        totflow += mk;
        totcost += co*mk;
        for (int i = T; i != S; i = pred[i].first) {
            cap[pred[i].second] -= mk;
            cap[pred[i].second^1] += mk;
        }
    }
    return make_pair(totflow, totcost);
}
```

4.7.2 Bowen

Running time $\mathcal{O}(FVE)$ if the pursued flow is F (in most cases, this should be faster). This algorithm only works if there are no negative cycles.

```
int dst[...], pre[...], pret[...];
bool spfa(){
    REP(i,0,FN) dst[i] = INF;
    dst[S] = 0;
    queue<int> que; que.push(S);
    while(!que.empty()){
        int x = que.front(); que.pop();
        for (int t : ra[x]){
            int y = kend[t], nw = dst[x] + cost[t];
            if(cap[t] > 0 && nw<dst[y]){
                dst[y] = nw; pre[y] = x; pret[y] = t; que.push(y);
            }
        }
    }
    return dst[T]!=INF;
}
// returns the maximum flow and the minimum cost for this flow
pair<ll,ll> solve(){
    ll totw = 0, totf = 0;
    while(spfa()){
        int minflow = INF;
        for (int x = T; x!=S; x = pre[x]){
            minflow = min(minflow, cap[pret[x]]);
        }
        for (int x = T; x!=S; x = pre[x]){
            cap[pret[x]] -= minflow;
            cap[pret[x]^1] += minflow;
        }
        totf += minflow;
        totw += minflow*dst[T];
    }
    return make_pair(totf, totw);
}
```

4.8 Hungarian/Kuhn–Munkres algorithm

Running time $\mathcal{O}(N^3)$. The following code returns the maximum weight of a matching of size `N1`. It assumes that there is such a matching.

```
const int INF = 1E9; // ∞: be careful to make this big enough but not too big!!!
int N1, N2; // Number of nodes in M1 and M2
int ni2[...]; // The node in M1 that is matched to a ∈ M2; -1 if unmatched
int praw[...], aa[...], bb[...], slack[...];
bool va[...], vb[...];
vector<PII> ad1[...]; // ad1[a]: Pairs (b,w) with b ∈ M2 that are connected to a ∈ M1
with an edge of weight w
bool find(int x){
    if(va[x]) return 0;
    va[x] = 1;
    for (PII t : ad1[x]) {
        int y = t.first;
        if(!vb[y] && aa[x]+bb[y]==t.second){
            vb[y] = 1;
            if(ni2[y]==-1 || find(ni2[y])) {
                praw[x]=t.second; ni2[y]=x; return 1;
            }
        }
        slack[y] = min(slack[y], aa[x]+bb[y]-t.second);
    }
    return 0;
}
int km(){
    FILL(bb,0); FILL(praw,0); FILL(ni2, -1);
    REP(i,0,N1){
        aa[i] = -INF;
        for (PII t : ad1[i])
            aa[i] = max(aa[i], t.second);
    }
    REP(x,0,N1){
        while(1){
            FILL(va,0); FILL(vb,0);
            REP(j,0,N2) slack[j] = INF;
            if(find(x)) break;
            int d = INF;
            REP(j,0,N2) if(!vb[j] && slack[j]<d) d = slack[j];
            REP(i,0,N1) if(va[i]) aa[i] -= d;
            REP(j,0,N2){
                if(vb[j]) bb[j] += d;
                else slack[j] -= d;
            }
        }
```

```
    }
    int ans = 0;
    REP(i,0,N1) ans += praw[i];
    return ans;
}
```

4.9 Strongly connected components

```
int N;
VI adj[...];
int tim;
int vis[...], low[...];
stack<int> st;
int comp[...]; // comp[i] is the number of the component containing node i
int compnr; // Number of components (numbered from 0 to compnr-1)

void visit(int i) {
    if (vis[i])
        return;
    tim++;
    vis[i] = low[i] = tim;
    st.push(i);
    for (int k : adj[i]) {
        visit(k);
        low[i] = min(low[i], low[k]);
    }
    if (low[i] == vis[i]) {
        while(true) {
            int k = st.top();
            st.pop();
            comp[k] = compnr;
            low[k] = 1E9;
            if (k == i)
                break;
        }
        compnr++;
    }
}
void solve() {
    tim = 0;
    compnr = 0;
    fill_n(vis, N, 0);
    REP(i,0,N)
        visit(i);
}
```

4.10 Articulation points and bridges

```
int N;
VI adj[...];
int tim;
int vis[...], low[...];
bool isart[...]; // isart[i] denotes whether i is an articulation point
vector<PII> bridges; // The bridges (pairs of nodes) (only in one direction)
void visit(int i, int p) {
    if (vis[i])
        return;
    tim++;
    vis[i] = low[i] = tim;
    int numch = 0, ho = tim;
    isart[i] = false;
    for (int k : adj[i]) {
        if (k != p) {
            if (!vis[k]) {
                visit(k, i);
                numch++;
                ho = min(ho, low[k]);
                if (low[k] >= vis[i])
                    isart[i] = true;
                if (low[k] > vis[i])
                    bridges.push_back(PII(i,k));
            } else
                ho = min(ho, vis[k]);
        }
    }
    low[i] = ho;
    if (p == -1)
        isart[i] = (numch >= 2);
}
void solve() {
    tim = 0;
    fill_n(vis, N, 0);
    bridges.clear();
    REP(i,0,N)
        visit(i, -1);
}
```

4.11 Euler paths

Finds a Euler path or Euler cycle in an undirected graph (if possible).

```
int N; // Number of nodes
int M; // Number of edges
vector<PII> adj[...]; // List of pairs (target, edge number); the edges have to be numbered
from 0 to M-1
int startadj[...];
bool visedge[...]; // ...should be at least the number of EDGES!!!
VI path;

void visit(int i) {
    int &s = startadj[i]; // Don't forget the "&"
    while(s < (int)adj[i].size()) {
        PII k = adj[i][s];
        if (visedge[k.second])
            s++;
        else {
            visedge[k.second] = true;
            visit(k.first);
        }
    }
    path.push_back(i);
```

```
}

// Returns whether there is a Euler path/cycle.
// If there is one, path will contain the nodes on a Euler path/cycle in the correct order.
// On a cycle, the starting point occurs at the beginning and at the end of path.
bool solve() {
    fill_n(startadj, N, 0);
    fill_n(visedge, M, false);
    path.clear();
    int au = 0;
    REP(i,0,N)
        au += adj[i].size()%2;
    if (au > 2) // use au > 0 if you are looking for Euler CYCLES
        return false;
    REP(i,0,N) {
        if ((au == 0 && adj[i].size()) || adj[i].size()%2) {
            visit(i);
            break;
        }
    }
    reverse(path.begin(), path.end());
    return (int)path.size() == M+1;
}
```

4.12 Orienting edges

Orients the edges of an undirected 2-connected graph in such a way that it becomes strongly connected.

```
int N;
VI adj[...]; // Adjacency list of the undirected graph
int hoe[...], par[...];

void dfs(int a) {
    for (int b : adj[a]) {
        if(hoe[b])
            continue;
        par[b] = a;
        hoe[b] = hoe[a] + 1;
        dfs(b);
    }
}

// An undirected edge ab should be oriented a → b if and only if
// par[b] == a || (par[a] != b && hoe[a] > hoe[b])
void solve() {
    REP(i,0,N)
        hoe[i] = par[i] = 0;
    hoe[0] = 1;
    dfs(0);
}
```

5 Geometry

5.1 Formulas

The area of a polygon with vertices $(x_0, y_0), \dots, (x_{n-1}, y_{n-1})$ is (up to sign!) $A = \frac{1}{2}\sum_{i=0}^{n-1}(x_i y_{i+1} - x_{i+1} y_i)$, where we reduce indices modulo n.
The x-coordinate of the center of mass of such a polygon is $\frac{1}{6A}\sum_{i=0}^{n-1}(x_i + x_{i+1})(x_i y_{i+1} - x_{i+1} y_i)$.
The x-coordinate of the circumcenter of the triangle $(0,0), (x_1, y_1), (x_2, y_2)$ is $\frac{y_2(x_1^2+y_1^2) - y_1(x_2^2+y_2^2)}{2(x_1 y_2 - y_1 x_2)}$.
Obtain the circumcenter of a general triangle by moving it!
The incenter of the triangle A, B, C is given by $\frac{A\cdot\overline{BC} + B\cdot\overline{CA} + C\cdot\overline{AB}}{\overline{BC}+\overline{CA}+\overline{AB}}$.

5.2 Fabian

5.2.1 General

```
struct pu {
    co x,y;
    pu(co a=0, co b=0) {x=a;y=b;}
};
pu operator-(const pu &a, const pu &b) {
    return pu(a.x-b.x,a.y-b.y);
}
// Not always necessary!
bool operator==(const pu &a, const pu &b) {
    return a.x == b.x && a.y == b.y;
}
pu operator*(co a, const pu &b) {
    return pu(a*b.x, a*b.y);
}

co kr(const pu &a, const pu &b) { // z component of the cross product a × b
    return a.x*b.y-b.x*a.y;
}
co kr(const pu &a, const pu &b, const pu &c) { // z component of the cross product (b − a) × (c − a)
    return kr(b-a,c-a);
}
// Intersection of the (infinite) lines a1a2 and b1b2 (if they aren't parallel).
// You obviously have to use floating point numbers, here!
pu inter(const pu &a1, const pu &a2, const pu &b1, const pu &b2) {
    return (1/kr(a1-a2,b1-b2))*(kr(a1,a2)*(b1-b2) - kr(b1,b2)*(a1-a2));
}
```

5.2.2 Convex hull

Finds the vertices of the convex hull (no points inside the edges of the convex hull). The points whose convex hull should be determined have to be *distinct*.

```
int N; // Number of points (≥ 1)
pu p[...]; // The points to determine the convex hull of
// The points will be reordered!
// Put the old indices somewhere (e.g., in struct pu) if you need them afterwards.
// h will contain the (new!) indices of the vertices of the convex hull in counterclockwise order
VI h;
bool swi(const pu &a, const pu &b) {
    pu as = a-p[0], bs = b-p[0];
```

```
    return kr(as, bs) > 0 || (kr(as,bs) == 0 &&
        (as.x < bs.x || (as.x == bs.x && as.y < bs.y)));
}
void solve() {
    REP(i,1,N)
        if (p[i].x < p[0].x || (p[i].x == p[0].x && p[i].y < p[0].y))
            swap(p[i], p[0]);
    sort(p+1, p+N, swi);
    h.clear();
    h.push_back(0);
    REP(i,1,N) {
        while((int)h.size() >= 2 &&
              kr(p[h[h.size()-2]], p[i], p[h[h.size()-1]]) >= 0)
            h.pop_back();
        h.push_back(i);
    }
}
```

5.2.3 Do the segments intersect?

```
// If a,b,c are collinear, this function returns whether c lies on the line segment [ab]
bool between(const pu &a, const pu &b, const pu &c) {
    return (c.x-a.x)*(c.x-b.x) <= 0 &&
           (c.y-a.y)*(c.y-b.y) <= 0;
}
bool gr(const pu &a1, const pu &a2, const pu &b1, const pu &b2) {
    co w1 = kr(b1-a1,a2-a1), w2 = kr(a2-a1,b2-a1);
    if (w1 == 0 && w2 == 0)
        return between(a1,a2,b1) || between(a1,a2,b2) ||
               between(b1,b2,a1) || between(b1,b2,a2);
    return (w1 >= 0 && w2 >= 0) || (w1 <= 0 && w2 <= 0);
}
// Returns whether the line segments [a1a2] and [b1b2] intersect
bool intersects(const pu &a1, const pu &a2, const pu &b1, const pu &b2) {
    return gr(a1,a2,b1,b2) && gr(b1,b2,a1,a2);
}
```

5.2.4 Does the point lie inside the polygon?

$$\text{Return value} = \begin{cases} 0, & \text{point outside} \\ 1, & \text{point inside} \\ 2, & \text{point on the border} \end{cases}$$

```
int N; // Number of nodes of the polygon
pu p[...]; // Nodes of the polygon

int inpolygon(const pu &a) {
    int num = 0;
    REP(i,0,N) {
        pu b = p[i]-a;
        pu c = p[(i+1)%N]-a;
        if (b.x == 0 && b.y == 0)
            return 2;
        if (b.y > c.y)
            swap(b,c);
        if ((b.x*c.x < 0 || b.y*c.y < 0) && kr(b,c) == 0)
            return 2;
        if (b.y < 0 && c.y >= 0 && kr(b,c) >= 0)
            num++;
    }
    return num%2;
}
```

5.3 Bowen

5.3.1 Points

```
struct pt2{
    double x, y;
    pt2(){}
    pt2(double _x, double _y){ x=_x; y=_y; }
    double len(){ return sqrt(x*x+y*y); }
    double dist(const pt2 &p){
        double dx = x-p.x, dy = y-p.y;
        return sqrt(dx*dx+dy*dy);
    }
    pt2 normalize() {   // must be non-zero vector!
        double s=sqrt(x*x+y*y);
        return pt2(x/s,y/s);
    }
    double dot(const pt2 &p) { return x*p.x+y*p.y; }
    double cross(const pt2 &p) { return x*p.y-y*p.x; }
    // cross return only value here
    pt2 rot(double th) { // around origin O
        return pt2(cos(th)*x-sin(th)*y, sin(th)*x+cos(th)*y);
    }
    // overload +,-,*,/ in the same way
    pt2 operator + (const pt2 &p) { return pt2(x+p.x,y+p.y); }
    pt2 operator - (const pt2 &p) { return pt2(x-p.x,y-p.y); }
};
int side(pt2 o, pt2 a, pt2 b){
    pt2 va = a-o, vb = b-o;
    double d = va.cross(vb);
    if(abs(d)<=EPS) return 0;
    return d>0?1:-1;
}
double angle(pt2 o, pt2 a, pt2 b){  // be careful of precision
    pt2 va = (a-o).normalize(), vb = (b-o).normalize();
    return acos(va.dot(vb));
}
bool inside(const pt2 &p, const vector<pt2> &polygon){
// sum of angles, this method is prone to precision error!
// shooting ray with integers is preferred!!!
    int sz = polygon.size();
    double ang = 0;
    REP(i,0,sz) ang += angle(p, polygon[i], polygon[(i+1)%sz]);
    return abs(abs(ang)-M_PI*2)<=EPS; // recommended EPS = 1E-9
}

struct pt3 {
```

```cpp
    double x,y,z;
    pt3(){}
    pt3(double _x, double _y, double _z){x=_x;y=_y;z=_z;}
    double dot(const pt3 &ano) { return x*ano.x+y*ano.y+z*ano.z; }
    pt3 cross(const pt3 &ano){
        double nx = y*ano.z-z*ano.y, ny = -(x*ano.z-z*ano.x),
        nz = x*ano.y-y*ano.x;
        return pt3(nx, ny, nz);
    }
    pt3 normalize(){
        double len = sqrt(x*x+y*y+z*z);
        return pt3(x/len, y/len, z/len);
    }
    // overload +,-,*,/ in the same way
    pt3 operator + (const pt3 &ano) { return pt3(x+ano.x, y+ano.y, z+ano.z); }
    pt3 operator - (const pt3 &ano) { return pt3(x-ano.x, y-ano.y, z-ano.z); }
    pt3 operator * (const double &val) { return pt3(x*val, y*val, z*val); }
    double len(){ return sqrt(x*x+y*y+z*z); }
};
double angle(pt3 &A, pt3 &B){    // be careful of precision
    return acos(A.normalize().dot(B.normalize()));
}

// rotate a 3D point A around a 3D vector OB by th,
// in counter-clockwise (right-hand) direction
pt3 rotate3(pt3 A, pt3 B, double th){
    pt3 e3 = B.normalize();
    double d = A.dot(e3);
    pt3 P = e3*d;
    pt3 e1 = (A-P).normalize();
    pt3 e2 = e3.cross(e1);
    double x1 = A.dot(e1), y1 = A.dot(e2);
    double xx,yy;
    xx = x1*cos(th)-y1*sin(th);
    yy = x1*sin(th)+y1*cos(th);
    return e1*xx+e2*yy+P;
}
```

5.3.2 Lines

```cpp
struct line2{
    double a, b, c; //ax+by+c=0;
    line2(){}
    line2(double _a, double _b, double _c){ a=_a; b=_b; c=_c; }
    line2(const pt2 &p1, const pt2 &p2){
        a = p2.y-p1.y; b = p1.x-p2.x; c = p2.x*p1.y-p1.x*p2.y;
    }
    pt2 intersect(const line2 &l){ // assume the lines are not parallel
        return pt2((b*l.c-c*l.b)/(a*l.b-b*l.a),
                   (a*l.c-c*l.a)/(b*l.a-a*l.b));
    }
    line2 perpline(const pt2 &p){ return line2(b,-a,a*p.y-b*p.x); }
    bool parallel(const line2 &l){ return abs(a*l.b-b*l.a)<=EPS; }
    double angle(const line2 &l){
        pt2 v1(a,b), v2(l.a,l.b);
        return acos(v1.dot(v2)/v1.len()/v2.len());
    }
    double dist(const pt2 &p){ // distance with sign (might be negative)
        return (a*p.x+b*p.y+c)/sqrt(a*a+b*b);
    }
};

struct seg2{
    pt2 a, b;
    seg2(pt2 _a, pt2 _b) { a=_a; b=_b; }
    double len(){ return a.dist(b); }
    pt2 project(const pt2 &p){
        pt2 v(b.x-a.x, b.y-a.y), vp(p.x-a.x, p.y-a.y);
        v = v.normalize();
        double d = vp.dot(v);
        return pt2(a.x+v.x*d, a.y+v.y*d);
    }
};
```

5.3.3 Convex hull

```cpp
bool operator< (const pt2 &p, const pt2 &q) {
    if(abs(p.x-q.x)<=EPS) return p.y < q.y;
    return p.x < q.x;
}
int N;
pt2 pts[...];
vector<pt2> hull;
void convexHull(){
    sort(pts, pts+N); // sort by increasing x, then y
    vector<pt2> hl[2];
    REP(t,0,2){
        int sn = t?-1:1;
        vector<pt2> &hu = hl[t];
        REP(i,0,N){
            pt2 c = pts[i];
            while(hu.size() >= 2 &&
                    side(hu[hu.size()-2], hu.back(), c) != sn)
                hu.pop_back();
            hu.push_back(c);
        }
    }
    for(int i=(int)hl[1].size()-2; i>=1; i--) hl[0].push_back(hl[1][i]);
    hull = hl[0]; // counterclockwise
}
```

6 Strings

6.1 Searching a word (Knuth-Morris-Pratt algorithm)

```cpp
int cont[...]; // At least 10 + maximum length of the word (string to search for)
char word[...]; // String to search for
char text[...]; // String to search inside
VI matches; // Will contain the places (starting indices) where the word occurs

void solve() {
    int A = strlen(word), B = strlen(text);
```

```
    cont[0] = -1;
    REP(i,1,A+1) {
        int pos = cont[i-1];
        while(pos != -1 && word[pos] != word[i-1])
            pos = cont[pos];
        cont[i] = pos+1;
    }
    int sp = 0, kp = 0;
    matches.clear();
    while(sp < B) {
        while(kp != -1 && (kp == A || word[kp] != text[sp]))
            kp = cont[kp];
        kp++;
        sp++;
        if(kp == A)
            matches.push_back(sp-A);
    }
}
```

6.2 Prefix tree

```
const int LETTERS = 60; // Number of different letters
int P; // Number of nodes in the prefix tree
char ltt[...]; // Character for this node
VI wend[...]; // Numbers of the words ending here
int adj[...][LETTERS]; // [node][character]: child node (-1 if non-existent)
vector<int> vadj[...]; // Numbers of the child nodes

int dec(char c) { // "character → number" mapping, for example:
    return c-'a';
}

void init() {
    wend[0].clear();
    fill_n(adj[0], LETTERS, -1);
    vadj[0].clear();
    P = 1;
}

// Add a word to the prefix tree and give it the number w
void add(const char *wort, int w) {
    int c = 0;
    for (int i = 0; wort[i]; i++) {
        int &cs = adj[c][dec(wort[i])]; // Don't forget the "&"
        if (cs == -1) {
            cs = P;
            vadj[c].push_back(P);
            ltt[P] = wort[i];
            wend[P].clear();
            fill_n(adj[P], LETTERS, -1);
            vadj[P].clear();
            P++;
        }
        c = cs;
    }
    wend[c].push_back(w);
}
```

6.3 Suffix array (with longest common prefixes)

Running time: $\mathcal{O}(N \log(N))$
The string has to end with a character whose value is smaller than all other characters in the string (the character $ is smaller than letters and digits, but bigger than whitespace). Just append such a character. The characters all have to be non-negative. If you want to handle multiple strings, concatenate them, separated by pairwise distinct characters smaller than all normal characters. If there are many strings, you should use `int str[...]` and shift the characters of the original word to make enough space for the special word-separator and end characters.
For a string s, let S_i be the suffix of s starting at character i (e.g., S_0 is the entire string). The rank of a suffix S_i is the number of suffixes smaller than S_i.

```
int N;  // number of characters in a string
// make all arrays larger than the number of characters in the string and the biggest character plus 10
int RA[...], SA[...], tmpRA[...], tmpSA[...], cnt[...];
// SA[i]: which suffix has rank i
// RA[i]: the rank of suffix S_i (the reverse of SA)
int lcp[...], phi[...];
// lcp: length of the longest common prefix of the suffixes of rank i-1 and i: S_SA[i-1] and S_SA[i]
// phi: the suffix that comes immediately before suffix i in the suffix array. That is, phi[SA[i]] = SA[i-1], phi[SA[0]] = -1
char str[...]; // the input string
void csort(int k){
    int cub = max(N, 300);  // take the maximum of the length of the strings and the number of unique chars
    FILL(cnt, 0);
    REP(i,0,N) cnt[i+k<N?RA[i+k]:0]++;
    REP(i,1,cub) cnt[i] += cnt[i-1];
    for(int i=cub-1;i>=1;i--) cnt[i] = cnt[i-1];
    cnt[0] = 0;
    REP(i,0,N) tmpSA[ cnt[SA[i]+k<N?RA[SA[i]+k]:0]++ ] = SA[i];
    REP(i,0,N) SA[i] = tmpSA[i];
}
void buildSA(){ // compute SA and RA
    REP(i,0,N){
        RA[i] = str[i];
        SA[i] = i;
    }
    int k = 1;
    while(k<N){
        csort(k); csort(0);      // cannot reverse order
        int r = 0;
        tmpRA[SA[0]] = 0;
        REP(i,1,N){
            if(RA[SA[i]]!=RA[SA[i-1]] || RA[SA[i]+k]!=RA[SA[i-1]+k])
                r++;
            tmpRA[SA[i]] = r;
        }
        REP(i,0,N) RA[i] = tmpRA[i];
        if(RA[SA[N-1]]==N-1) break;
        k <<= 1;
    }
}
```

```
void buildLCP(){ // compute lcp and phi (optional); call this after buildSA()
    int L=0;
    phi[SA[0]] = -1;
    REP(i,1,N) phi[SA[i]] = SA[i-1];
    REP(i,0,N){
        if(phi[i]==-1){ lcp[RA[i]]=0; continue; }
        while(str[i+L]==str[phi[i]+L]) L++;
        lcp[RA[i]] = L;
        if(L) L--;
    }
}
```

6.4 Finding longest palindromes (Manacher's algorithm)

```
char str[MAXN]; // first read string into this array
int pl[MAXN*2]; // pl[i] is the length of the longest palindrome centered at i/2 (where 0 means
"at the first character", 1/2 means "between the first and the second character", and so on)
void manacher(){
    int L = strlen(str);
    int C=-1; // C must be set to -1
    memset(pl, 0, sizeof(pl));
    REP(k,0,2*L-1){
        if(2*C-k>=0 && k+pl[2*C-k]/2*2-(2*C-k)%2<C+pl[C]/2*2-C%2){
            pl[k] = pl[2*C-k];
        }else{
            int i;
            if(2*C-k>=0){
                pl[k] = min(pl[2*C-k], (C+pl[C]/2*2-C%2-k+1)*2/2);
                i = C+pl[C]/2*2-C%2+2;
            }else{
                pl[k] = k%2==0;
                i = k+(k%2?1:2);
            }
            C = k;
            while(2*C-i>=0 && str[i/2]==str[(2*C-i)/2]) {
                i+=2; pl[C]+=2;
            }
        }
    }
}
```

6.5 Automata

Constructing a nondeterministic automaton M from a regular expression A
Idea: Recursively construct the automaton.

- **If** $A = \emptyset$: Add start and end node and connect the start node to itself via an ϵ-transition.
- **If** $A = a$: Add start and end node and connect them via an a-transition.
- **If** $A = BC$: Let M_A be the union of M_B and M_C.
  Add an ϵ-transition $\text{end}(M_B) \to \text{start}(M_C)$.
  Let $\text{start}(M_A) = \text{start}(M_B)$ and
  $\text{end}(M_A) = \text{end}(M_C)$.
- **If** $A = B|C$: Let M_A be the union of M_B and M_C.
  Add a new start and a new end node. Add ϵ-transitions:
  $\text{start}(M_A) \to \text{start}(M_B), \text{start}(M_C)$
  $\text{end}(M_B), \text{end}(M_C) \to \text{end}(M_A)$.
- **If** $A = B^*$: Add a new start and a new end node. Add ϵ-transitions:
  $\text{start}(M_A) \to \text{start}(M_B)$
  $\text{end}(M_B) \to \text{start}(M_A)$
  $\text{end}(M_A) \to \text{end}(M_A)$

Useful properties of such automata:

- Every state has at most two outgoing transitions.
- If a state has two outgoing transitions, then both are ϵ-transitions.
- A state is the end if and only if it has no outgoing transitions.
- There is exactly one end state.

If necessary, you can eliminate ϵ-transitions in the following way: Compute the transitive closure of the ϵ-transitions and add an edge $Z_i \xrightarrow{a} Z_k$ for all $Z_i \xrightarrow{a} Z_j \xrightarrow{\epsilon} Z_k$. Mark the start state as end state if there is an ϵ-transition between them.

6.6 Prefix Automaton (Aho-Corasick algorithm)

Search multiple words simultaneously. You first have to create a prefix tree containing the words to search for.
Running time $\mathcal{O}(S + N + M)$ if S is the sum of the lengths of the words to search for (not the number of nodes in the prefix tree!), N is the length of the text to search and M is the number of matches.

```
int suf[...], preend[...], dep[...], par[...];

void initaho() {
    queue<int> qu;
    qu.push(0);
    par[0] = -1;
    dep[0] = 0;
    while(!qu.empty()) {
        int i = qu.front();
        qu.pop();
        if (i == 0)
            suf[i] = preend[i] = -1;
        else {
            int s = suf[par[i]];
            while(s != -1 && adj[s][dec(ltt[i])] == -1)
                s = suf[s];
            suf[i] = s == -1 ? 0 : adj[s][dec(ltt[i])];
            preend[i] = wend[suf[i]].size() ? suf[i] : preend[suf[i]];
        }
        for (int k : vadj[i]) {
            par[k] = i;
            dep[k] = dep[i]+1;
            qu.push(k);
        }
    }
}
```

```
vector<PII> matches; // All pairs (a,b) such that word number b starts at position a

void search(const char *text) {
    matches.clear();
    int c = 0;
    for (int i = 0; text[i]; i++) {
        while(c != -1 && adj[c][dec(text[i])] == -1)
            c = suf[c];
        c = c == -1 ? 0 : adj[c][dec(text[i])];
        int t = c;
        while(t != -1) {
            for (int w : wend[t])
                matches.push_back(PII(i-dep[t]+1, w));
            t = preend[t];
        }
    }
}
```

6.7 Suffix Automaton

$O(N)$ time suffix automaton construction.

```
#define MAXN 20005 // double the size of N for MAXN!
int L, tl, par[MAXN], ch[MAXN][26], ml[MAXN], cc[MAXN]; // following the par pointers to get all the ac states
void init(){
    L = tl = ml[0] = 0;
    par[0] = -1;
    FILL(ch, 0);
}
void extend(int c, int len){
    int p = tl, np = ++L;
    ml[np] = len;
    cc[np] = len;
    for(;p!=-1 && !ch[p][c]; p=par[p]) ch[p][c] = np;
    tl = np;
    if(p==-1) par[np] = 0;
    else{
        if(ml[ch[p][c]]==ml[p]+1) par[np] = ch[p][c];
        else{
            int q = ch[p][c], r = ++L;
            par[r] = par[q];     // copy q to r
            cc[r] = cc[q];
            memcpy(ch[r], ch[q], sizeof(int)*26);
            ml[r] = ml[p]+1;
            par[q] = par[np] = r;
            for(;p!=-1 && ch[p][c]==q; p=par[p]) ch[p][c] = r;
        }
    }
}
```

7 Miscellaneous

7.1 Floyd's cycle finding algorithm

Let $f : \mathbb{Z} \to \mathbb{Z}$ be a function. The following algorithm returns the smallest numbers $a \geq 0$ (the cycle start) and $b \geq 1$ (the cycle length) such that $f^{(a)}(s) = f^{(a+b)}(s)$ where $f^{(t)}$ denotes the t-fold composition of f with itself.
Running time: $\mathcal{O}(a + b)$

```
pair<int,int> floyd(int s){
    int x=f(s), y=f(f(s)), cycst=0, cyclen=1;
    while(x!=y){ x=f(x); y=f(f(y)); }
    y = s; while(x!=y){ x=f(x); y=f(y); cycst++; }
    y = f(x); while(x!=y){ y=f(y); cyclen++; }
    return make_pair(cycst, cyclen);    // cycle start, cycle length
}
```

7.2 Segment tree (increase an interval and ask for the maximum in an interval)

```
int NSEG; // Number of fields (with indices 0,...,NSEG-1)
int maxin[...]; // initialize to 0, WARNING: Should have at least size 2·2^k + 10 with 2^k > N and k ∈ Z+
int off[...]; // initialize to 0, WARNING: Should have at least size 2·2^k + 10 with 2^k > N and k ∈ Z+

void relax(int i) {
    maxin[i] += off[i];
    off[2*i+1] += off[i];
    off[2*i+2] += off[i];
    off[i] = 0;
}

// add(a,b,v) increases the values at a,...,b-1 by v
void add(int a, int b, int v, int i=0, int s=0, int e=NSEG) {
    if (b <= s || e <= a)
        return;
    if (a <= s && e <= b) {
        off[i] += v;
        return;
    }
    relax(i);
    add(a, b, v, 2*i+1, s, (s+e)/2);
    add(a, b, v, 2*i+2, (s+e)/2, e);
    maxin[i] = max( // Change this if you want the minimum.
        maxin[2*i+1]+off[2*i+1],
        maxin[2*i+2]+off[2*i+2]);
}

// query(a,b) returns the maximum of the values at a,...,b-1
int query(int a, int b, int i=0, int s=0, int e=NSEG) {
    if (b <= s || e <= a)
        return -1E9; // Change this if you want the minimum (should be a neutral element).
    if (a <= s && e <= b)
        return off[i]+maxin[i];
```

```
    relax(i);
    return max(  // Change this if you want the minimum.
        query(a, b, 2*i+1, s, (s+e)/2),
        query(a, b, 2*i+2, (s+e)/2, e));
}
```

7.3 Binary Indexed Tree

For 2D BIT, use two nested loop in the same way as 1D. The returned result is the prefix sum of the upper-left sub-matrix.

```
// Initialize this to 0:
int ba[...];  // ...should be at least the maximum index plus 10

// Returns a_0 + ... + a_i
int get(int i) {
    i++;
    int e = 0;
    for (; i; i -= i&-i)
        e += ba[i];
    return e;
}

// Increases a_i by v
void add(int i, int v) {
    i++;
    for (; i <= ...; i += i&-i)  // Use the maximum index for ...
        ba[i] += v;
}
// find the k-th smallest element, s = ba
int findkth(int *s, int k){
    int e = 0, cnt = 0;
    for(int i=20;i>=0;i--){
        int b = 1<<i;
        if(e+b>=... || cnt+s[e+b]>=k) continue;
        e += b;
        cnt += s[e];
    }
    return e;
    // this is the largest e where sum1:e<k,
    // however, as we shift *s by 1 (k++), it should be e+1-1 = e
}
```

7.4 Heavy-Light Tree Decomposition

```
int N;
int w[MAXN],sz[MAXN],pref[MAXN],lb[MAXN],lbl;  // node weight, subtree size, preferred child, relabeled id
int par[MAXN],top[MAXN],dep[MAXN];  // parent, path top, node depth, indexes are in relabeled node ids
int val[1<<20];  // segment tree, make large enough
int dfs(int x, int p){
    sz[x] = 1;
    int t = adj[x], maxsz = 0;
    while(t!=-1){
        int y = lists[t].id;
        if(y==p) {
            t = lists[t].next;
            continue;
        }
        int s = dfs(y, x);
        sz[x] += s;
        if(s>maxsz){ maxsz = s; pref[x] = y; }
        t = lists[t].next;
    }
    return sz[x];
}
void dfs2(int x, bool st, int lbp, int d){
    lb[x] = lbl++;
    par[lb[x]] = lbp;
    dep[lb[x]] = d;
    if(st) top[lb[x]] = lb[x];
    else top[lb[x]] = top[lbp];
    if(pref[x]!=-1) dfs2(pref[x], 0, lb[x], d+1);
    int t = adj[x];
    while(t!=-1){
        int y = lists[t].id;
        if(lb[y]!=-1){
            t = lists[t].next;
            continue;
        }
        dfs2(y,1,lb[x],d+1);
        t = lists[t].next;
    }
}
// import standard segment tree code here
// ...
int solve(int x, int y){    // answer query for path (a,b)
    int a, ans = 0;
    while(top[x]!=top[y]){
        if(dep[top[x]] < dep[top[y]]) swap(x,y);
        a = querySeg(0, 0, N-1, top[x], x);
        ans = max(a, ans);
        x = top[x];
        if(dep[x] >= dep[top[y]]) x = par[x];
    }
    if(dep[x] > dep[y]) swap(x,y);
    a = querySeg(0, 0, N-1, x, y);
    ans = max(ans, a);
    return ans;
}
void init(){    // read graph before init
    FILL(lb, -1); FILL(w, 0); FILL(pref, -1);
    dfs(0, -1);
    lbl = 0;
    dfs2(0, 1, -1, 0);
    FILL(val, 0);
}
```

7.5 Splay Tree

```
int L, rt, tch[MAXN][2], tsz[MAXN], par[MAXN], tc[MAXN], stk[MAXN];
```

```
int key[MAXN], val[MAXN];
bool rev[MAXN];
// L: insert count
// rt: the current splay root
// par: parent of a node
// tch: tree children
// tsz: size of subtree (num of nodes)
// tc: type of node (left/right/root), tc is used to determine a root (not par!)
// key, val: (key,value) pair
// nodes start from 1, node 0 is for aux use

// splay has a DUMMY node with INF key to avoid being empty
void init(){
    L = 1; tsz[0] = 0;
    rt = 1; tc[rt] = 2; // dummy root to avoid empty tree
    tch[rt][0] = tch[rt][1] = 0;
    key[rt] = INF; // root has INF that will never be touched
    val[rt] = 1;
}
inline void update(int x){
    tsz[x] = 1+tsz[tch[x][0]]+tsz[tch[x][1]];
}
inline void relax(int x){    // propagate the change to children, called in splay
    if(rev[x]){
        swap(tch[x][0], tch[x][1]);
        tc[tch[x][0]] = 0;
        rev[tch[x][0]] ^= 1;
        tc[tch[x][1]] = 1;
        rev[tch[x][1]] ^= 1;
        rev[x] = 0;
    }
}
void rotate(int x){
    int c = tc[x], y = par[x], z = tch[x][1-c];
    tc[x] = tc[y];
    par[x] = par[y];
    if(tc[x]!=2) tch[par[x]][tc[x]] = x;
    tc[y] = 1-c;
    par[y] = x;
    tch[x][1-c] = y;
    if(z){
        tc[z] = c;
        par[z] = y;
    }
    tch[y][c] = z;
    update(y);
}

void splay(int x, bool relaxParents=false){
    // if you find or insert x, the parents are already relaxed
    // set relaxParents to true only if you directly splay this node, usually in LCT
    if(relaxParents){
        int stksz = 0;
        stk[stksz++] = x;
        for(int i=x; tc[i]!=2; i=par[i]) stk[stksz++] = par[i];
        for(int i=stksz-1; i>=0; i--) relax(stk[i]);
    }
    while(tc[x]!=2){
        int y = par[x];
        if(tc[x]==tc[y]) rotate(y);
        else rotate(x);
        if(tc[x]==2) break;
        rotate(x);
    }
    update(x);
    rt = x;
}
int findkth(int x, int k){   // find the k-th node, if no such node, return 0
    if(!x) return 0;
    relax(x);
    if(tsz[tch[x][0]]+1==k) return x;
    if(tsz[tch[x][0]]>=k) return findkth(tch[x][0], k);
    return findkth(tch[x][1], k-tsz[tch[x][0]]-1);
}
void remove(int x){ // remove node x
    splay(x);
    int xl = tch[x][0], xr = tch[x][1];
    if(xl==0){   // no left child!
        tc[xr] = 2;
        rt = xr;
        return;
    }
    tc[xl] = 2; // set root
    int y = findkth(xl, tsz[xl]);    // find the largest node
    splay(y);
    tch[y][1] = xr;
    tc[xr] = 1;
    par[xr] = y;      // concat xr to y's right child
}
void insertKey(int ky){
    int x = rt, px=-1;
    while(x){
        relax(x);
        if(key[x]==ky){ val[x]++; break; }
        px = x;
        x = tch[x][ky>key[x]];
    }
    if(!x){
        x = ++L;
        tch[x][0] = tch[x][1] = 0;
        tsz[x] = 1; key[x] = ky; val[x] = 1;
        rev[x] = 0;

        int c = ky>key[px];
        tch[px][c] = x;
        par[x] = px;
        tc[x] = c;
    }
    splay(x);    // this splay would update the parents
}
void removeKey(int ky){ // assume existence;
    int x = rt;
```

```
    while(1){
        relax(x);
        if(key[x]==ky) break;
        x = tch[x][ky>key[x]];
    }
    if(val[x]>1) {
        val[x]--;
        splay(x);
    }else remove(x);
}
int findmin(){
    int x = rt;
    while(x) {
        relax(x);
        if(tch[x][0]) x = tch[x][0];
        else break;
    }
    splay(x);
    return key[rt];
}
```

7.6 Leftist Tree

$O(\log N)$ time merging two heaps.

```
#define MAXN 100005
int tch[MAXN][2], val[MAXN], ss[MAXN];
// tch: tree children, left=0, right=1
// ss: balance factor
// element index is the same as the tree node index, starting from 1
int merge(int a, int b){     // merge two leftist trees with roots a, b (must be root!)
    if(a==-1) return b;
    if(b==-1) return a;
    if(val[a] > val[b]) swap(a,b);
    tchr[b] = merge(a, tchr[b]);
    if(ss[tchr[b]] > ss[tchl[b]]) swap(tchl[b], tchr[b]);
    ss[b] = tchr[b]==-1? 0: ss[tchr[b]]+1;
    return b;
}
int removeRoot(int a){   // remove a leftist tree root (a must be root!)
    int r = merge(tch[a][0], tch[a][1]);
    tch[a][0] = tch[a][1] = 0; // now a is a single node tree
    return r;
}
int insert(int a, int b){    // a is a node to be inserted, b is a leftist tree root
    val[a] = ... // set value for a
    tchl[a] = tchr[a] = 0;
    return merge(a, b);
}
void init(){
    ss[0] = -1; // -1+1=0
    REP(i,1,1+N) {
        root[i] = i;
        val[i] = ... // read in node values
    }
    FILL(tch, 0);
}
```

7.7 Link-Cut Tree

```
int N;
// splay tree
int par[MAXN], val[MAXN], tsum[MAXN], tch[MAXN][2], tc[MAXN], stk[MAXN], add[MAXN];
bool rev[MAXN];
int pathpar[MAXN];   // LCT
void init(){
    FILL(add,0);
    FILL(tsum, 0); FILL(val, 0);
    REP(i,1,N+1){
        tc[i] = 2;
        tch[i][0] = tch[i][1] = 0;   // one node each splay tree
        pathpar[i] = 0;
    }
}
void relax(int x){}
void update(int x){}
void rotate(int x){
    int y; // copy splay tree...
    par[x] = par[y];
    pathpar[x] = pathpar[y]; // add this line
    if(tc[x] != 2) tch[par[x]][tc[x]] = x;
    // ...
    update(y);
    update(x);   // add this to be safe in LCT
}
void splay(int x){
    int stksz = 0;
    stk[stksz++] = x;
    for(int i=x; tc[i]!=2; i=par[i]) stk[stksz++] = par[i];
    for(int i=stksz-1; i>=0; i--) relax(stk[i]);
    while(tc[x]!=2){
        int y = par[x];
        if(tc[x]==tc[y]) rotate(y);
        else rotate(x);
        if(tc[x]==2) break;
        rotate(x);
    }
}
int expose(int x){ // returns the lowest node last exposed
    int y = 0;
    while(x){
        splay(x);
        int r = tch[x][1];
        tc[r] = 2;   // cut right subtree off
        pathpar[r] = x;

        tch[x][1] = y;   // connect to new path
        par[y] = x; tc[y] = 1;
        update(x);
        y = x; x = pathpar[x];
    };
    return y;
}
int findroot(int x){
```

```
    expose(x);
    splay(x);
    while(tch[x][0]) x = tch[x][0];
    splay(x);
    return x;
}
void join(int x, int y){ // x is root of some tree, setroot(x) before join
    //setroot(x);
    pathpar[x] = y;
    expose(x);
}
void setroot(int x){
    expose(x);
    splay(x);
    rev[x] ^= 1;
    pathpar[x] = 0;
}
```

7.8 2-SAT

$(v_0 \vee \neg v_1) \wedge (v_3 \vee v_2) \wedge (\neg v_1 \vee \neg v_0)$ with variables v_0, v_1, v_2, v_3 is encoded as:

```
A = 3
V = 4
```

v_0	condsig[0][0] = 1 condvar[0][0] = 0	v_3	condsig[1][0] = 1 condvar[1][0] = 3	$\neg v_1$	condsig[2][0] = 0 condvar[2][0] = 1
$\neg v_1$	condsig[0][1] = 0 condvar[0][1] = 1	v_2	condsig[1][1] = 1 condvar[1][1] = 2	$\neg v_0$	condsig[2][1] = 0 condvar[2][1] = 0

```
// ... has to be greater than the number of conditions and TWICE the number of variables

int A, V; // Number of conditions (a or b) that have to be fulfilled, number of variables
bool condsig[...][2]; // The "sign" of the part of the condition (false means "not")
int condvar[...][2]; // Number of the variable in the part of the condition
bool solution[...]; // A possible assignment of variables; not necessary if the question is just WHETHER there is a solution

VI adj[...], radj[...];
bool vis[...];
VI order;
int comp[...];
int compnr;
VI incomp[...], adjcomp[...]; // not necessary if the question is just WHETHER there is a solution
bool viscomp[...], isset[...]; // not necessary if the question is just WHETHER there is a solution

void visitcomp(int i) { // not necessary if the question is just WHETHER there is a solution
    if (viscomp[i])
        return;
    viscomp[i] = true;
    for (int k : adjcomp[i])
        visitcomp(k);
    isset[i] = true;
    for (int k : incomp[i])
        solution[k/2] = (isset[comp[incomp[i][0]^1]]+k)%2;
}

void findsolution() { // not necessary if the question is just WHETHER there is a solution
    REP(i,0,compnr) {
        incomp[i].clear();
        adjcomp[i].clear();
        viscomp[i] = false;
        isset[i] = false;
    }
    REP(i,0,2*V) {
        incomp[comp[i]].push_back(i);
        for (int k : adj[i])
            adjcomp[comp[i]].push_back(comp[k]);
    }
    REP(i,0,compnr)
        visitcomp(i);
}

VI *cadj;
void dfs(int i) {
    if(vis[i])
        return;
    vis[i] = true;
    comp[i] = compnr;
    for (int k : cadj[i])
        dfs(k);
    order.push_back(i);
}

bool solve() { // Returns whether all conditions are simultaneously satisfiable
    REP(i,0,2*V) {
        adj[i].clear();
        radj[i].clear();
    }
    REP(i,0,A) {
        int v0 = 2*condvar[i][0]+condsig[i][0];
        int v1 = 2*condvar[i][1]+condsig[i][1];
        adj[v0^1].push_back(v1);
        radj[v1].push_back(v0^1);
        adj[v1^1].push_back(v0);
        radj[v0].push_back(v1^1);
    }
    fill_n(vis, 2*V, false);
    order.clear();
    cadj = adj;
    REP(i,0,2*V)
        dfs(i);
    fill_n(vis, 2*V, false);
    compnr = 0;
    cadj = radj;
    for (int i = 2*V-1; i >= 0; i--)
        if (!vis[order[i]]) {
            dfs(order[i]);
            compnr++;
```

```
        }
    REP(i,0,V)
        if (comp[2*i] == comp[2*i+1])
            return false;
    findsolution(); // not necessary if the question is just WHETHER there is a solution
    return true;
}
```

7.9 Solve a linear system of equations

```
int N; // Number of variables
int M; // Number of equations
double mat[...][...]; // [equation][variable]
double vec[...], sol[...];
int pivot[...];
// Returns whether mat*sol = vec has a solution sol
bool solve() {
    REP(k,0,M) {
        double ma = 0; // Maybe 1e-10 ?; Over F_p, this is not necessary
        int p = -1;
        REP(i,0,N) {
            if (ma < abs(mat[k][i])) { // Over F_p, use if mat[k][i]%p
                ma = abs(mat[k][i]); // Over F_p, this is not necessary
                p = i;
            }
        }
        pivot[k] = p;
        if (p == -1)
            continue;
        REP(l,k+1,M) {
            double f = mat[l][p]/mat[k][p]; // Over F_p, use (precalculated?) inverses
            REP(j,0,N)
                mat[l][j] -= f*mat[k][j]; // Over F_p, take this modulo p
            vec[l] -= f*vec[k]; // Over F_p, take this modulo p
        }
    }
    fill_n(sol, N, 0);
    for (int k = M-1; k >= 0; k--) {
        if (pivot[k] == -1) {
            if (abs(vec[k]) > 1e-10) // Adjust the tolerance; over F_p, use vec[k] != 0
                return false;
            continue;
        }
        double rest = vec[k];
        REP(i,0,N)
            rest -= sol[i]*mat[k][i];
        sol[pivot[k]] = rest/mat[k][pivot[k]]; // Over F_p: use (precalculated?) inverses and take this modulo p, be careful with negative numbers!
    }
    return true;
}
```

7.10 Fast Fourier transform

```
typedef complex<double> comp;

int S;
vector<comp> a, b, ei;

void ffth(int s, int r, int w) {
    if (s == S) {
        b[w] = a[r];
        return;
    }
    int nh = S/s/2;
    ffth(2*s, r, w);
    ffth(2*s, r+s, w+nh);
    REP(k,0,nh) {
        comp t1 = b[k+w], t2 = b[k+w+nh]*ei[k*s];
        b[k+w] = t1 + t2;
        b[k+w+nh] = t1 - t2;
    }
}

// For given a_0,...,a_{S-1} (where S should be a power of 2), calculates b_s = sum_{r=0}^{S-1} a_r e^{-2 pi i s r/S} for s = 0,...,S-1
void fft() {
    b.resize(S);
    ei.resize(S);
    comp e1 = polar(1.,-2*M_PI/S); // depending on the time limit, you might want to precalculate this
    ei[0] = 1;
    REP(i,1,S)
        ei[i] = ei[i-1]*e1;
    ffth(1,0,0);
}

// For vectors p and q of lengths P and Q, returns a vector r such that r_s = sum_{0<=t<=s} p_t q_{s-t} for s = 0,...,P+Q-1, where p and q are extended by 0 if needed.
vector<comp> convolution(const vector<comp> &p, const vector<comp> &q) {
    S = 1;
    while(S < (int)p.size() + (int)q.size())
        S *= 2;
    a.assign(S,0);
    copy(p.begin(), p.end(), a.begin());
    fft();
    vector<comp> c = b;
    a.assign(S,0);
    copy(q.begin(), q.end(), a.begin());
    fft();
    REP(i,0,S)
        a[i] = b[i]*c[i];
    fft();
    vector<comp> r(S);
    REP(i,0,S)
        r[i] = b[(S-i)%S]/(double)S;
    return r;
}
```

7.11 Hashing and generating random numbers

The following are pairs (p, a) of a prime p and a primitive root a modulo p:
$(1\,000\,000\,007, 234\,234\,234)$ and $(1\,000\,000\,009, 987\,654\,321)$ and $(999\,999\,937, 171\,317\,131)$

7.12 Alpha-Beta pruning

```
// GET get a piece's integer representation from the board
// SET set a piece's integer representation to the board
// util is the utility function of the game, write util function first
// call solve(bd, -INF, INF, player)
int solve(int bd, int alpha, int beta, bool player){
    int pc = !player?1:2, u;
    if(util(bd,u)) return u;
    int mx,my;
    REP(i,0,4) REP(j,0,4){ // this should loop over all moves
        int b = GET(i,j,bd);
        if(b) continue;
        int nbd = bd + SET(i,j,pc);
        int val = solve(nbd, alpha, beta, !player);
        if(!player){
            if(val>=beta) return val;
            if(val>alpha){ alpha=val; mx=i; my=j; if(val==1) return 1; }
        }else{
            if(val<=alpha) return val;
            if(val<beta){ beta=val; mx=i; my=j; if(val==-1) return -1; }
        }
    }
    wm = make_pair(mx,my); // the best move
    return !player?alpha:beta;
}
```

8 Java

8.1 Template

```
import java.io.*;
import java.util.*;
import java.util.regex.*;
import java.math.*;

public class javastuff {
    public static void main(String[] args) throws Exception {
    }
}
```

8.2 BufferedReader

```
BufferedReader in = new BufferedReader(new InputStreamReader(System.in));
while(true) {
    String s = in.readLine(); // No \n at the end of the string
    if (s == null) // EOF
        break;
}
```

8.3 Scanner

```
Scanner in = new Scanner(System.in);
while(in.hasNext()) {
    in.next(); // String
    in.nextInt();
    in.nextLong();
    in.nextDouble();
    in.nextBigInteger();
    in.nextInt(16); // Hexadecimal
}
```

8.4 Printing

```
System.out.print("abc");
System.out.println();
System.out.print("fuenf = "+5+" ");
System.out.println(10);
System.out.println(Integer.toString(210,16)); // d2 hexadecimal
System.out.println(Integer.toString(210,16).toUpperCase()); // D2 hexadecimal
System.out.format("%09d %s %X %c\n", 123, "hi", 255, 'a');
```

8.5 String

```
String s = "  fdsfa asf=d s   dsfdsf   ";
s.length();
s.charAt(1); // f
s.split(" "); // {"", "", "fdsfa", "asf=d", "in", "", "", "dsfdsf"}
s.split(" +"); // {"", "fdsfa", "asf=d", "in", "dsfdsf"}
s.substring(7,9); // "as"
s.equals("dsfdfas"); // false (WARNING: Do not use ==)
s.startsWith("  fd"); // true
s.replace('f','_'); // " _ds_a as_=d s   ds_ds_   "
s.replace("fd","_#"); // " _#sfa asf=d s   ds_#sf   "
s.replaceAll("\\s+","_"); // "_fdsfa_asf=d_s_dsfdsf_"
s.replaceFirst("\\s+","_"); // "_fdsfa asf=d s   dsfdsf   "
s.matches("[a-z ]*"); // true
Pattern p = Pattern.compile("(.*)\\=([a-z ]*)");
Matcher m = p.matcher(s); // tries to match the complete string s
m.matches(); // true
m.group(2); // "d s   dsfdsf   "
```

8.6 Arrays

```
int a[] = {6,5,7};
Arrays.sort(a); // inplace sort: a = {5,6,7}
int l = a.length;
int mat[][] = new int[10][10];
BigInteger matb[][] = new BigInteger[10][10];
BigInteger b = matb[1][2]; // null
for (int i = 0; i < 10; i++)
    for (int j = 0; j < 10; j++)
        matb[i][j] = BigInteger.valueOf(0);
```

8.7 ArrayList

```java
ArrayList<Integer> v = new ArrayList<Integer>();
v.add(6);
v.add(5);
v.add(7);
Collections.sort(v);
Collections.sort(v, new Comparator<Integer>() {
    public int compare(Integer a, Integer b) {
        if (a%2 == b%2)
            return 0;
        if (a%2 == 0)
            return -1;
        else
            return +1;
    }
});
v.get(2); // 7
v.set(1,2); // v[1] = 2
v.size(); // 3
v.remove(v.size()-1); // pop_back
v.clear();
```

8.8 Queue

```java
Deque<Integer> q = new ArrayDeque<Integer>();
q.addLast(5);
q.addFirst(7);
q.getFirst();
q.removeFirst();
q.removeLast();
```

8.9 TreeSet

```java
TreeSet<Integer> s = new TreeSet<Integer>();
s.add(6);
s.add(5);
s.add(8);
s.first(); // 5 (smallest element)
s.last(); // 8 (largest element)
s.ceiling(7); // 8 (smallest element ≥ 7)
s.ceiling(9); // null
s.floor(7); // 6 (largest Element ≤ 7)
s.contains(5); // true
s.remove(5);
s = new TreeSet<Integer>(new Comparator<Integer>() {
    public int compare(Integer a, Integer b) {
        if (a%2 == b%2)
            return 0;
        if (a%2 == 0)
            return -1;
        else
            return +1;
    }
});
```

8.10 TreeMap

```java
TreeMap<Integer,Integer> m = new TreeMap<Integer,Integer>();
m.put(5,6);
m.put(10,11);
m.containsKey(5);
m.get(5);
m.remove(5);
```

8.11 Math

```java
double pi = Math.PI;
Math.sin(90*pi/180);
Math.abs(-2.0);
Math.atan2(2,1); // atan2(y,x)
Math.exp(2);
Math.sqrt(2);
Math.floor(2.3);
Math.round(2.3);
Math.max(7,3);
Math.random(); // double 0 ≤ x < 1
```

8.12 BigInteger

```java
BigInteger a = new BigInteger("123456789123456789123456789");
BigInteger b = BigInteger.valueOf(11111);
a.add(b);
a.multiply(b);
a.subtract(b);
a.divide(b);
a.remainder(b); // could be negative
a.mod(b); // non-negative
a.pow(5);
a.isProbablePrime(20); // wrong with probability ≤ 1 - 2^-20
a.max(b); // max(a,b)
a.min(b); // min(a,b)
a.abs(); // |a|
a.gcd(b); // gcd(a,b)
a.modInverse(b); // a^-1 mod b
a.modPow(BigInteger.valueOf(100000),b); // a^100000 mod b
b.intValue(); // 12345 as int
b.longValue(); // 12345 as long
b.doubleValue(); // 12345 as double
a.equals(b); // false
b.equals(BigInteger.valueOf(11111)); // true
b.equals(11111); // false!!!
a.compareTo(b); // -1, a < b;  0, a = b;  1, a > b
a.toString(16); // hexadecimal
```

Task	People-Status	Description
A		
B		
C		
D		
E		
F		
G		
H		
I		
J		
K		
L		
M		