```
   000   00000    0000   011101010
   010   01110    0111   000011000     Rafael Paulino
   010   01110   00 00   00
   001   010 00  01 01   001110000     Renato Paes Leme
   010   010 01  01 00   010011100
   010   010 00 01  01   01            Thiago Siqueira
   011   010  0100  00   000110010
   010   010  0010  00   000011101     coach: Claudia Justel

       ATLANTICO's Notebook
```

------- CONTENT -------

```
/**************************************************************************
*  C++ Model                                                              *
**************************************************************************/
#include  <cstdio>
#include  <cstdlib>
#include  <string>
#include  <cmath>
#include  <inttypes.h>
#include  <ctype.h>
#include <algorithm>
#include <utility>
#include <iostream>
#include <vector>

using namespace std;

#define TRACE(x...)
#define PRINT(x...) TRACE(printf(x))
#define WATCH(x) TRACE(cout << #x" = " << x << "\n")

#define tr(it,s) for(typeof(s.begin())it=s.begin();it!=s.end();++it)
#define rep(i,n) for(int i=0; i<n; ++i)

const int INF = 0x3F3F3F3F;
const int NULO = -1;
const double EPS = 1e-10;

inline int cmp(double x, double y = 0, double tol = EPS) {
  return (x <= y + tol) ? (x + tol < y) ? -1 : 0 : 1; }

/**************************************************************************
*  Compiling Script                                                       *
**************************************************************************/
#!/bin/sh
clear
rm -f $1.out
if (g++ -o $1 $1.cpp -Wall -pedantic -lm -g) then
   echo "### COMPILOU ###"
   if !(./$1 < $1.in > $1.out) then
      echo "### RUNTIME ERROR ###" >> $1.out
   fi
   less $1.out
fi
```

```
/*****************************************************************
*  STL Utils                                                     *
*****************************************************************/
Leitura de String:
#include <sstream>
istringstream ins; ins.str(s); ins >> a >> b; //IN
ostringstream outs; outs << a << b; s = outs.str(); //OUT

upper_bound(v.begin(), v.end(), key);//justafter the last element
found
lower_bound(v.begin(), v.end(), key);//first element found, or, int
the case the element is not found, in the first position it could be
inserted without violating the order ( first element greater or
equal key )
binary_search(v.begin(), v.end(), key);
swap(a,b); //swaps a and b

random_shuffle(v.begin(), v.end());
next_permutation(v.begin(), v.end());

//Heap without priority change
priority_queue<ii,vector<ii>, greater<ii> > Q;
Q.push<ii(priority,key)>; //insert element
int u = Q.top().second; Q.pop(); //remove element

//Heap with priority change ( priorities in v[] )
struct comp{
        inline bool operator() (const int i, const int j) {
                return (v[i] != v[j]) ? (v[i] < v[j]) : (i < j);
        }
};
set<int, comp> s;
s.erase(x); v[x]=new_priority; s.insert(x); //priority change
v[x] = priority; s.insert(x); //insert element
int u = *s.begin(); s.erase(*s.begin()); //remove element

#define contains(s,x) ( s.find(x) != s.end() ) //set contains element

fill(v.begin(), v.end(), key); //fills the structure with key
remove(v.begin(), v.end(), val); //removes all elements with val key
replace(v.begin(),v.end(), val, new_val);
reverse(v.begin(),v.end());
find(v.begin(), v.end(), key); //returns first iterator to key
find_if(v.begin(), v.end(), pred);//where pred is bool pred(T key)
unique(v.begin(), v.end()); //eliminates duplicates

//String Functions ( s is a string )
s.find(string pattern, int pos=0); // find a pattern in string
starting from position zero. returns index of first element if
pattern is found and a large number if not found
s.substr(int pos, int length);
//Other Utils
bool cmp_eq(double x, double y) { return cmp(x, y) == 0; }
bool cmp_lt(double x, double y) { return cmp(x, y) < 0; }
```

```
/*****************************************************************
*  Limits                                                        *
*****************************************************************/
      tipo         |bits|      mínimo .. máximo
  -----------------+----+-----------------------------------
char               |  8 |           0 .. 127
signed char        |  8 |        -128 .. 127
unsigned char      |  8 |           0 .. 255
short              | 16 |     -32.768 .. 32.767
unsigned short     | 16 |           0 .. 65.535
int                | 32 |  -2 × 10**9 .. 2 × 10**9
unsigned int       | 32 |           0 .. 4 × 10**9
int64_t            | 64 | -9 × 10**18 .. 9 × 10**18
uint64_t           | 64 |           0 .. 18 × 10**18
float              | 32 |  10**38     with precision  6
double             | 64 |  10**308    with precision 15
long double        | 80 |  10**19.728 with precision 18

//Number of primes until n:
pi(10**5)   = 9.592
pi(10**6)   = 78.498
pi(10**7)   = 664.579
pi(10**8)   = 5.761.455
pi(10**9)   = 50.847.534
[n / ln(n) < pi(n) < 1.26 * n / ln(n)]

//Pascal triangle elements:
C(33,  16)  =               1.166.803.110  [int limit]
C(34,  17)  =               2.333.606.220  [unsigned int limit]
C(66,  33)  = 7.219.428.434.016.265.740    [int64_t limit]
C(67,  33)  = 14.226.520.737.620.288.370   [uint64_t limit]

//Fatorial

12! =               479.001.600 [(unsigned) int limit]
20! = 2.432.902.008.176.640.000 [(unsigned) int64_t limit ]

Some 'medium' primes:
8837 8839 8849 8861 8863 8867 8887 8893 8923 8929 8933
8941 8951 8963 8969 8971 8999 9001 9007 9011 9013 9029
9041 9043 9049 9059 9067 9091 9103 9109 9127 9133 9137
Cousins: (229,233),(277,281) Twins: (311,313),(347,349),(419,421)
Some not so small primes:
80963,263911,263927,567899,713681,413683,80963
37625701, 236422117, 9589487273, 9589487329, 694622169483311

/*****************************************************************
*  Nice Functions                                                *
*****************************************************************/

//Euler phi for n = p_1 ^ b_1 * ... * p_k ^ b^k
phi(n) = prod ( p_i - 1 )*( p_i ^ ( b_i - 1 ), i=1...k )
       = n * prod ( 1 - ( 1 / p_i ) )
Euler theorem: if gcd(a,n) = 1 then a ^ phi(n) ~ 1 ( mod n )
```

```
//Binomial coefficient C[n][k] = n! / ( k! * (n-k)! )
int C[TAM][TAM];
void calc_pascal() {
  memset(C, 0, sizeof(C));
  for (int i = 0; i < TAM; i++) {
    C[i][0] = C[i][i] = 1;
    for (int j = 1; j < i; j++)
      C[i][j] = C[i-1][j-1] + C[i-1][j];
  }
}

//Catalan Numbers
Catalan(n) = (1/(n+1))*C[2n][n] = (2n)! / ( n! * (n+1)! )
Catalan(n+1) = (2*(2n+1)/(n+2)) * Catalan(n)
             = sum_{i=0..n} Catalan(i)*Catalan(n-i) with Catalan(0)=1

Catalan(n) is: (i) expressions with n pairs of parenthesis correctly
matched, (ii) ways of performing a binary associative operation on
a[0]*...*a[n] ( n+1 factors ), (iii) number of rooted binary trees
with n+1 leaves (if leaves are labeled and the operation is
commutative, we have an (n+1)! factor), (iv) number of monotonic
paths along the edges of a grid with n × n square cells, which do
not pass above the diagonal, (v) the number of different ways a
convex polygon with n + 2 sides can be cut into triangles by
connecting vertices with straight lines

//Stirling Number of the first kind
x(x-1)...(x-n+1) = sum_{k=0..n} s(n,k) x^k
s(n+1,k) = s(n,k-1)-n*s(n,k) --> s(n,0)=delta(n), s(0,1)=0, s(1,1)=1
The absolute value of the Stirling number of the first kind, s(n,k),
counts the number of permutations of n objects with exactly k orbits
(equivalently, with exactly k cycles).

//Stirling numbers of the second kind
(x)_n = x(x-1)...(x-n+1) --> x^n = sum_{k=0..n} S(n,k) (x)_k
S(n,k) = S(n-1,k-1) + k*S(n-1,k) --> S(n,1) = 1, S(n,n) = 1
S(n,k)%2 == ! ( (n-k) & ((k-1)/2) );
S(n,n-1) = C[n][2] = n(n-1)/2
S(n,k) gives the number of possible rhyming schemes for n lines
using k unique rhyming syllables.

//Bell Number
B(n+1) = sum_{k=0..n} C[n][k] B(k) = sum_{k=1...n} S(n,k)
if p is prime B(p+n) ~ B(n) + B(n+1) ( mod p ) [ Touchard ]
Bn is the number of partitions of a set of size n. A partition of a
set S is defined as a set of nonempty, pairwise disjoint subsets of
S whose union is S.


//Mobius mi function ( n = prod ( p[i] ^ e[i] , i =1...k ) )
mi(1) = 1, mi(n) = (-1)^k if e[i]=1 for i=1...k, mi(n)=0 otherwise
sum_{d|n}mi(d) = 1 if n==1 and 0 otherwise
Inversion formula: f(n)=sum_{d|n}g(d) <=> g(n)=sum_{d|n}mi(d)f(n/d)
sigma(n)=sum_{d|n}d = prod( (p[i]^(e[i]-1) - 1)/(p[i]-1) ) //sigma
```

//NIM Game
Rules of the Game of Nim: There are n piles of coins. When it is a player's turn he chooses one pile and takes at least one coin from it. If someone is unable to move he loses (so the one who removes the last coin is the winner). If the piles have size n1,...,nk, then it is a losing position for the player whose turn it is if and only if n1 xor n2 xor .. xor nk = 0.

//Pick's Theorem
Pick's theorem provides a simple formula for calculating the area A of this polygon in terms of the number i of interior points located in the polygon and the number b of boundary points placed on the polygon's perimeter: A = i + (b/2) - 1

```
/*****************************************************************
* Calculates all the primes from 2 to n                          *
*****************************************************************/
int prime[200000]; int nprimes;
void find_primes(int n) {
  nprimes = 1; prime[0] = 2; bool is_prime;
  for(int k=3;k<=n;k+=2) {
    is_prime = true;
    for(int j=0; j<nprimes && prime[j]*prime[j] <= k;++j)
      if (k%prime[j] == 0) {is_prime = false; break;}
    if ( is_prime ) prime[nprimes++] = k;
  }
}

/*****************************************************************
* Factorizes n. We get n = prod(p[i]^e[i])                       *
* Requires to know primes from 2 to sqrt(n)                      *
*****************************************************************/
int p[20000]; int e[20000]; int k;
void factor(int n) {
  k = 0;
  for(int j=0; prime[j]*prime[j] <= n; ++j) if ( n%prime[j]==0 ) {
    p[k] = prime[j]; e[k] = 0;
    while ( n%prime[j]==0 ) { n/=prime[j]; e[k]++; }
    ++k;
  }
  if (n!=1) { p[k]=n; e[k] =1; ++k;}
}

/*****************************************************************
* Calculates all positive divisors of prod (p[i]^e[i])           *
* Requires the factorization of the number                       *
*****************************************************************/
int divisor[20000]; int ndivisors;
void divisors() {
  ndivisors=1; divisor[0] = 1; int r;
  rep(i,k) {
    r = ndivisors;
    rep(j,e[i]) rep(a,r) divisor[ndivisors++] = p[i]*divisor[a+j*r];
  }
}
```

```cpp
int gcd(int x, int y) { return y ? gcd(y, x % y) : abs(x); }

uint64_t lcm(int x, int y) {
  if (x && y) return abs(x) / gcd(x, y) * uint64_t(abs(y));
  else return uint64_t(abs(x | y));
}

/**********************************************************************
* Bezout Theorem: finds bezout=(a,b) such that a*x+b*y=gcd(x,y)     *
**********************************************************************/
typedef pair<int, int> bezout;
bezout find_bezout(int x, int y) {
   if (y == 0) return bezout(1, 0);
   bezout u = find_bezout(y, x % y);
   return bezout(u.second, u.first - (x/y) * u.second);
}

/**********************************************************************
* Solves the equation a*x=b (mod m) or returns -1 if the solution   *
* does not exit. It can be used to invert a number mod m  ( m > 1 ) *
**********************************************************************/
int mod(int x, int m) { return x % m + ( (x < 0) ? m : 0 ); } //m!=1
int solve_mod(int a, int b, int m) {
   if (m < 0) return solve_mod(a, b, -m);
   if (a < 0 || a >= m || b < 0 || b >= m)
     return solve_mod(mod(a, m), mod(b, m), m);
   bezout t = find_bezout(a, m);
   int d = t.first * a + t.second * m;
   if (b % d) return -1;
   else return mod(t.first * (b / d), m);
}
int inverse_mod(int x, int m) {return solve_mod(x,1,m);}

/**********************************************************************
* Chinese Remainder Theorem: Calculates x so that x ~ b[i] mod m[i] *
* does not require m[i] and m[j] to be co-prime. Returns solution   *
* mod lcm(m[i]) if it exists or -1 if it doesn't.                   *
**********************************************************************/
int b[20000]; int m[20000]; int bsize;
int chinese_remainder() {
  int b0=b[0], m0=m[0], d;
  for(int i=1; i<bsize; ++i) {
    d = gcd(m0,m[i]);
    if ( d!=1 && mod( b0 - b[i], d ) != 0 ) return -1;
    b0 = b0 + ( solve_mod ( m0/d, (b[i]-b0)/d, m[i]/d ) )*m0;
    m0 = lcm(m0,m[i]);
    b0 = mod (b0,m0);
  }
  return b0;
}
```

```
/*****************************************************************
* Simplex and Linear Systems: You should fill the tableau T. Let *
* A be an m x n matrix, so the tableau will be (with b[m] and c[n]):*
*                          [ -f   |   c[n]   ]                   *
*            T[m+1][n+1] = [----------------]                   *
*                          [ b[m] |  A[m][n] ]                   *
*****************************************************************/
#define irep(i,i0,n) for(int i = i0; i <= n; ++i)
const int nmax = 100, mmax = 100; int n,m;
double T[nmax+mmax+1][mmax+1]; // tableau
double x[nmax+1]; /*solution 1..n*/ int B[mmax+1];/*basic vars*/

void pivot( int l, int j ) {
  rep(k,n+1) if (k!=j) T[l][k] /= T[l][j]; T[l][j] = 1;
  rep(i,m+1) if (i!= l) {// for simplex
  //irep(i,1,m) if (i!= l) { // for linear systems
    rep(k,n+1) if (k!=j) T[i][k] -= T[l][k]*T[i][j];
    T[i][j] = 0;
  }
}

/*****************************************************************
* Simplex: solves min c^t x s.t. Ax=b, x>=0. If you already know *
* a basis, write it in B[m] and just call simplex(),otherwise,   *
* you need to call two_phase_simplex(). If rankA < m, the value  *
* of m will be decreased by the two_phase procedure.             *
*****************************************************************/
double simplex(bool &unbounded) {
  irep(i,1,m) pivot( i, B[i] );
  bool optimal = false; int j; unbounded = false;
  while ( !(optimal || unbounded) ) {
    optimal = true;
    for(j=1; j<=n; ++j) if ( cmp(T[0][j])< 0)
      {optimal = false; break;}
    if (optimal) return -T[0][0];
    double theta = INF; int leave;
    for(int i=1; i<=m; ++i) if ( cmp(T[i][j]) > 0 )
      if ( cmp ( T[i][0], theta * T[i][j] ) < 0 ) {
        leave = i;
        theta = T[i][0] / T[i][j];
      }
      else if ( ( cmp ( T[i][0], theta * T[i][j] ) == 0 ) &&
                                ( B[i] < B[leave] ) ) {
        leave = i; theta = T[i][0] / T[i][j];
      }
    if (theta == INF) {unbounded = true; return -T[0][0];}
    pivot(leave,j); B[leave] = j;
  }
}

double two_phase_simplex( bool &unbounded, bool &infeasible ) {
  unbounded = infeasible = false;
  irep(i,1,m) if ( T[i][0] < 0 ) rep(j,n+1) T[i][j] *= -1;
  double c[n+1]; rep(j,n+1) c[j] = T[0][j];
  int N = n; n += m;
```

```
  irep(j,1,n) T[0][j] = 0;
  irep(j,1,m) {
    T[0][N+j] = 1; B[j] = N+j;
    irep(i,1,m) T[i][N+j] = ( i == j );
  }
  double csi = simplex(unbounded);
  if ( cmp(csi,0) > 0 ) {
    infeasible = true; return -T[0][0];
  }
  irep(i,1,m) if ( B[i] > N ) {
    bool redundant = true;
    // driving artificial variables out of the basis
    irep(j,1,N) if ( cmp(T[i][j]) != 0 ) {
      pivot(i,j); B[i] = j;
      redundant = false; break;
    }
    if ( redundant ) {
      rep(j,n+1) T[i][j] = T[m][j];
      B[i] = B[m]; --m; --i;
    }
  }
  n = N; rep(j,n+1) T[0][j] = c[j];
  return simplex(unbounded);
}

void get_solution() { //get solution to simplex
  irep(i,0,n) x[i] = 0; irep(i,1,m) x[B[i]] = T[i][0];
}

/*********************************************************************
* Linear Systems: solve Ax=b. Returns 1 if a solution was found.    *
* From D we get det(A). Solution is stored in x[1...n] ( n = m )    *
*********************************************************************/
#define abs(x) (x>=0 ? x : -x)
double D;
bool solve_linear_system() {
  D = 1; irep(j,1,m) T[0][j] = j;
  irep(i,1,m)
  {
    int p = i;
    irep(k,i+1,m) if ( cmp ( abs( T[k][i] ), abs( T[p][i] ) ) > 0 )
      p = k;
    if (p!=i) rep(j,n+1) swap(T[i][j], T[p][j]);
    if (cmp(T[i][i])==0) {
      p = i;
      irep(k,i+1,m) if ( cmp ( abs( T[i][k] ), abs( T[i][p] ) ) > 0 )
        p = k;
      if (p!=i) rep(j,m+1) swap(T[j][i], T[j][p]);
    }
    D*=T[i][i];
    if ( cmp(T[i][i])==0 & cmp(T[i][0])!=0 ) return false;
    else if ( cmp(T[i][i])!=0 ) pivot(i,i);
  }
  return true;
}
```

```
void get_solution() { irep(j,1,m) x[(int)T[0][j]] = T[j][0]; }

/**********************************************************************
* Invert a matrix A[m][m] - store the matrix in the tableau and the *
* identity in T[1..m][m+1...2m], make n = 2*m and call solve_linear *
* _system(). Get the inverse matrix at T[1..m][m+1...2m]            *
**********************************************************************/

/**********************************************************************
* This other code doesn't suppose m == n. After the execution, if   *
* possible, m is the rank of the matrix. m is the number of lines   *
* and n the number of columns ( variables ).                        *
**********************************************************************/
bool solve_linear_system() {
  irep(j,1,n) T[0][j] = j;
  irep(i,1,m<?n) {
    int p = i;
    irep(k,i+1,m) if (cmp(abs( T[k][i]), abs(T[p][i]))>0 ) p = k;
    if (p!=i) rep(j,n+1) swap(T[i][j], T[p][j]);
    if (cmp(T[i][i])==0) {
      p = i;
      irep(k,i+1,n) if (cmp(abs( T[i][k]), abs(T[i][p]))>0 ) p = k;
      if (p!=i) rep(j,m+1) swap(T[j][i], T[j][p]);
    }
    if ( cmp(T[i][i])==0 & cmp(T[i][0])!=0 ) return false;
    else if ( cmp(T[i][i])!=0 ) pivot(i,i);
    else {
      rep(j,n+1) swap(T[i][j], T[m][j]);
      --i; --m;
    }
  }
  irep(i,(m<?n)+1,m) if (cmp(T[i][0])!=0) return false;
  if (m>n) m = n;
  return true;
}

void get_solution() {
  irep(i,1,n) x[i] = 0;
  irep(j,1,m) x[(int)T[0][j]] = T[j][0];
}

/**********************************************************************
* Some bit tricks: WARNING: It is safer to use "unsigned" for       *
* this type of job                                                  *
**********************************************************************/

x & ~(x - 1) //only the lowest 1 bit remains on
__builtin_ctz(x) //count trailing zeros. so:
__builtin_ctz(x)+1 //lowest index with bit 1 ( don't call with x=0 )
__builtin_clz(x) //cound leading zeros
__builtin_popcount(x)//number of 1 bits
(i & (i << 1)) //check if there are adjacent bits

for(int x=0; x<(1<<n); ++x) //all subsets of {0,...,n-1}
```

```
for(int x=s; x!=0; x = (x-1)&s ) //all non-empty subsets of set s

//Iterate through all k-element subsets of {0, 1, … N-1}
int s = (1 << k) - 1;
while (!(s & 1 << N)) {
    // do stuff with s
    int lo = s & ~(s - 1); int lz = (s + lo) & ~s;
    s |= lz; s &= ~(lz - 1); s |= (lz / lo / 2) - 1;
}


/*********************************************************************
* Rational Numbers                                                   *
*********************************************************************/
class Rational {
  public:
  long p, q;
  Rational () {}
  Rational (long _p, long _q)  {
    long z = gcd (abs (_p), abs(_q));
    if (_q < 0) z = -z; p = _p / z; q = _q / z;
  }
  Rational (long _p): p(_p), q(1) {}
  Rational (int _p): p((long) _p), q(1) {}
  static long gcd (long a, long b) {
    while (b > 0) { long t = a % b; a = b; b = t; }
    return a;
  }
  operator string() const {
    ostringstream s; s << p << "/" << q; return s.str();
  }

  friend bool operator== (const Rational &A, const Rational &B);
  friend bool operator!= (const Rational &A, const Rational &B);
  friend bool operator< (const Rational &A, const Rational &B);
  friend Rational operator+ (const Rational &A, const Rational &B);
  friend Rational operator- (const Rational &A, const Rational &B);
  friend Rational operator* (const Rational &A, const Rational &B);
  friend Rational operator/ (const Rational &A, const Rational &B);
  friend ostream& operator <<(ostream& o, const Rational& A);
};


bool operator== (const Rational &A, const Rational &B) {
  if (A.p != B.p) return false; if (A.q != B.q) return false;
  return true;
}

bool operator!= (const Rational &A, const Rational &B) {
  if (A.p != B.p) return true; if (A.q != B.q) return true;
  return false;
}

bool operator< (const Rational &A, const Rational &B) {
  return ((A.p * B.q) < (A.q * B.p));
```

```
}

Rational operator+ (const Rational &A, const Rational &B) {
  long z = Rational::gcd(A.q, B.q);
  return Rational ( A.p*(B.q/z)+B.p*(A.q/z), A.q*(B.q/z) );
}

Rational operator- (const Rational &A, const Rational &B) {
  long z = Rational::gcd(A.q, B.q);
  return Rational ( A.p*(B.q/z)-B.p*(A.q/z), A.q*(B.q/z) );
}

Rational operator* (const Rational &A, const Rational &B) {
  return Rational ( A.p * B.p, A.q * B.q );
}

Rational operator/ (const Rational &A, const Rational &B) {
  return Rational ( A.p * B.q, A.q * B.p );
}

ostream& operator <<(ostream& o, const Rational& A) {
  return o << (string) A;
}



/*****************************************************************
* String Matching - Polynomial Hashing ( Rabin-Karp ) - the vector  *
* match contains first index of every occurence of pattern in text  *
*****************************************************************/
vector<int> text, pattern; int N = 4; //number of hashs
int p[4] = {0x5b56ea1b, 0x3d4c834d, 0x1197dd8b, 0x7d310bfd };
int q[4] = {0x9e6fe013, 0x4b86d85, 0xb8f8e223, 0xea11a955 };//q=p^-1
//string T; rep(i,T.size()) text.push_back( T[i] - ' ' );

void rabin_karp(vector<int> &match) {
  int n = text.size(), m = pattern.size(); match.clear();
  bool matched;
  int e[N]; rep(j,N) { e[j] = 1; rep(i,m-1) e[j] *= p[j]; }
  if ( n < m ) return;
  int H[N], h[N]; rep(j,N) H[j] = h[j] = 0;
  rep(j,N) rep(i,m) H[j] = H[j]*p[j] + pattern[i]; //mod 2^32
  rep(j,N) rep(i,m) h[j] = h[j]*p[j] + text[i]; //mod 2^32

  matched = true; rep(j,N) if (H[j] != h[j]) {matched=false; break;}
  if ( matched ) match.push_back(0);

  for(int i=m;i<n;++i) {
    rep(j,N) h[j] = p[j]*(h[j]-e[j]*text[i-m]) + text[i];
    matched = true;
    rep(j,N) if (H[j] != h[j]) {matched=false; break;}
    if ( matched ) match.push_back(i-m+1);
  }
}
```

```
/*************************************************************************
* String Matching - Knuth-Morris-Pratt ( before calling KMP with        *
* a pattern, you need to call build_prefix_function()                   *
*************************************************************************/
const int MMAX = 10000; vector<int> text, pattern; int pi[MMAX+1];

void build_prefix_function() {
  int m = pattern.size(); pi[0] = 0; int k = 0;
  for(int q=1; q<m; ++q) {
    while (k>0 && pattern[k] != pattern[q]) k = pi[k-1];
    if ( pattern[k] == pattern[q] ) ++k;
    pi[q] = k;
  }
}

void knuth_morris_pratt(vector<int> &match) {
  int n = text.size(); int m = pattern.size();
  int q = 0; // number of characters matched
  rep(i,n) {
    while (q>0 && pattern[q] != text[i]) q = pi[q-1];
    if ( pattern[q] == text[i] ) ++q;
    if ( q == m ) { match.push_back(i-m+1); q = pi[q-1]; }
  }
}

/*************************************************************************
* Segment Tree - call len[1] and borders[1] to know                      *
* the total segment length and the total borders .                       *
*************************************************************************/
typedef pair<int,int> ii;
long T[4*NMAX]; // segment tree
int len[4*NMAX], borders[4*NMAX], br[4*NMAX], bl[4*NMAX];
int N, off, r; //off = offset, r = log2(off)
int Y[ NMAX ];

void initialize(int _N) { // _N is number os segments (not endpoints)
  N = _N; off = 1; r = 0; while ( off < N ) { off <<= 1; ++r; }
  rep(i,2*off) T[i] = len[i] = borders[i] = br[i] = bl[i] = 0;
}

void update(int node, int b, int e) {
  if (T[node]) {
    len[node] = Y[e+1]-Y[b]; // or just E-B+1;
    borders[node] = 2; br[node] = bl[node] = 1;
  } else if ( node < off ) {
    len[node] = len[2*node] + len[2*node+1];
    borders[node] = 2*borders[2*node] + 2*borders[2*node+1]
                    - 2*br[2*node]*bl[2*node+1];
    bl[node] = bl[2*node]; br[node] = bl[2*node+1];
  } else {
    len[node] = borders[node] = br[node] = bl[node] = 0;
  }
}
```

```
void add(int B, int E, int node=1, int b=0, int e=off-1) {
  if ( e < B || E < b ) return; // or B==E ?
  else if ( B <= b && e <= E ) T[node]++;
  else  {
    add(B,E,2*node,b,(b+e)/2);
    add(B,E,2*node+1,1+(b+e)/2,e);
  }
  update(node,b,e);
}

void erase(int B, int E, int node=1, int b=0, int e=off-1)
{
  if ( e < B || E < b ) return; // or B==E ?
  else if ( B <= b && e <= E ) T[node]--;
  else {
    erase(B,E,2*node,b,(b+e)/2);
    erase(B,E,2*node+1,1+(b+e)/2,e);
  }
  update(node,b,e);
}

/**********************************************************************
* Rectangle Union                                                     *
**********************************************************************/
int x[NMAX][2], y[NMAX][2]; int nrect;
typedef pair<int,int> event;

long union_area() {
  if (nrect==0) return 0;
  vector< event > E; int m = 0;
  rep(i,nrect) {
    E.push_back( event(x[i][0],i) );
    E.push_back( event(x[i][1],~i) );
    Y[m++] = y[i][0]; Y[m++] = y[i][1];
  }
  sort(E.begin(), E.end()); sort(Y, Y+m);
  m = unique(Y, Y+m)-Y;
  int last = E[0].first, ans = 0;
  initialize(m-1);
  rep(i,2*nrect) {
    int k = E[i].second; bool in = (k>=0); if (!in) k = ~k;
    ans += ( E[i].first - last ) * len[1];
    int a = lower_bound(Y,Y+m, y[k][0]) - Y;
    int b = lower_bound(Y,Y+m, y[k][1]) - Y;
    if (in) add(a,b-1); else erase(a,b-1);
    last = E[i].first;
  }
  return ans;
}
```

```
/*************************************************************************
* Segment Tree for RangeMinimumQueries query( 1, 0, off-1, i, j );  *
* with complexity <O(n), O(log N)>                                      *
*************************************************************************/
long A[ NMAX ]; // values
long M[ 4 * NMAX ]; // segment tree
int N, off, r; //off = offset, r = log2(off)

void initialize(int node, int b, int e) {
  if ( b == e ) M[ node ] = b;
  else {
    initialize( 2*node, b, (b+e)/2 );
    initialize( 2*node+1, 1+(b+e)/2, e );
    if ( A[ M[2*node] ] <=  A[ M[2*node+1] ] )
      M[node] = M[2*node];
    else
      M[node] = M[2*node+1];
  }
}

void pre_processing() {
  off = 1; r = 0;
  while ( off < N ) { off <<= 1; ++r; }
  rep(i,N) M[ off+i ] = A[i];
  initialize( 1, 0, off-1);
}

int query( int node, int b, int e, int i, int j ) {
  if ( e < i || j < b ) return -1;
  else if ( i <= b && e <= j ) return M[node];
  else {
    int p1 = query( 2*node, b, (b+e)/2, i, j );
    int p2 = query( 2*node+1, 1+(b+e)/2, e, i, j);
    if (p1 == -1) return p2;
    else if (p2 == -1) return p1;
    else if (A[p1] <= A[p2])
      return p1;
    else
      return p2;
  }
}

/*************************************************************************
* Binary Search: evaluate the least index between lo and hi          *
* ( including those ) for which p(x) is true. ( p=[00..00111...1] ) *
*************************************************************************/
int binary_search(int lo, int hi, bool p(int) ) {
  int mid;
  while (lo < hi) {
    mid = lo + (hi-lo)/2;
    if ( p(mid) == true )
      hi = mid;
    else
      lo = mid+1;
  }
```

```
  if ( p(lo) == false ) {/* p(x) is false for all x in S! */  }
  return lo;  // lo is the least x for which p(x) is true
}


/*************************************************************
* Disjoint Sets (union-find) with path-compression and rank-union   *
*************************************************************/
int n; int pi[nMax]; int rank[nMax]; int size[nMax];

void initialize(int _n) {
  n = _n;
  for(int i=0; i<n; i++) { pi[i] = i; rank[i] = 0; size[i] = 1; }
}

int FindSet(int x) {
  if (x!=pi[x]) pi[x] = FindSet(pi[x]); //with path compression
  return pi[x];
}

void Union(int x, int y) {
  x = FindSet(x); y = FindSet(y);
  if (x==y) return;
  if (rank[x] > rank[y]) { //union by rank
    pi[y] = x; size[x] += size[y];
  } else {
    pi[x] = y; size[y] += size[x];
    if (rank[x] == rank[y]) rank[y]++;
  }
}

/*************************************************************
* Exact Cover - based on Knuth's Dancing Links use it by building   *
* ExactCover problem(m); where m is the number of columns then add  *
* the rows ( a deque with elements from 1 to m ). Then, call        *
* problem.solve() and all the solutions will be printed. If you     *
* want to do something with the solutions, modify print().          *
*************************************************************/

class ExactCover {
  public:
  int L[tam], R[tam], U[tam], D[tam], C[tam], P[tam];
  int S[nMax]; int O[nMax]; int next;
  int m; //number of columns
  int n; //number of rows

  ExactCover(int _m)
  {
    n = 0; m = _m;
    R[0] = 1; L[0] = m; U[0] = D[0] = C[0] = -1;
    for(int i=1; i<=m; i++) {
      L[i] = i-1; R[i] = i+1; U[i] = D[i] = i; S[i] = 0;
    }
    R[m] = 0; next = m+1;
  }
```

```
void addRow(deque<int> &row) {//integers from 1 to m
  int size = row.size(); n++;
  for(int i=next; i<next+size; i++)
      { L[i] = i-1; R[i] = i+1; P[i] = n; }
  L[next] = next+size-1; R[next+size-1] = next;
  for(int i=0; i<size; i++) {
    D[next+i] = row[i]; U[next+i] = U[row[i]];
    D[U[row[i]]] = next+i; U[row[i]] = next+i;
    C[next+i] = row[i]; S[row[i]]++;
  }
  next+= size;
}

void cover(int c) {
  L[R[c]] = L[c]; R[L[c]] = R[c];
  for(int i=D[c]; i!=c; i=D[i]) {
    for(int j=R[i]; j!=i; j = R[j]) {
      U[D[j]] = U[j]; D[U[j]] = D[j]; S[C[j]]--;
    }
  }
}

void uncover(int c) {
  for(int i=U[c]; i!=c; i=U[i]) {
    for(int j=L[i]; j!=i; j = L[j]) {
      S[C[j]]++; U[D[j]] = j; D[U[j]] = j;
    }
  }
  L[R[c]] = c; R[L[c]] = c;
}

int chooseColumn() {
  int c,s = n+1;
  for(int j=R[0]; j!=0; j = R[j]) {
    if (S[j]<s) {
      c = j; s = S[j];
    }
  }
  return c;
}

void print(int k) { //columns in each row
/*
  for(int i=0; i<k; i++) {//each row
    cout << C[O[i]] << " "; //this is the answer
    for(int j = R[O[i]]; j!=O[i]; j = R[j]) {
      cout << C[j] << " "; //this is the answer
    }
    cout << endl;
  }
  */
  //row numbers:
  for(int i=0; i<k; i++) cout << P[O[i]] << " ";
```

```
    cout << endl;
  }

  void search(int k=0) {
    if (R[0] == 0) { print(k); return; }
    int c = chooseColumn();
    cover(c);
    for(int r = D[c]; r!=c; r = D[r]) {
      O[k] = r;
      for(int j = R[r]; j!=r; j = R[j]) cover(C[j]);
      search(k+1);
      r = O[k]; c = C[r];
      for(int j = L[r]; j!=r; j = L[j]) uncover(C[j]);
    }
    uncover(c);
    return;
  }
};


/**************************************************************************
* Find all roots of a polynomial. If you already know some root,          *
* just call ruffini with it. You can also make a binary search            *
* to find a root and call ruffini.                                        *
**************************************************************************/
#include <complex>
typedef complex<double> cdouble;

int cmp(cdouble x, cdouble y = 0, double tol = EPS) {
   return cmp(abs(x), abs(y), tol); //needs comp for double
}

struct poly {
  cdouble p[NMAX+1]; int n;
  poly(int n = 0): n(n) { memset(p, 0, sizeof(p)); }
  cdouble& operator [](int i) { return p[i]; } //p(x)=sum (p[i]x^i)

  poly operator ~() { //derivate
    poly r(n-1);
    for (int i = 1; i <= n; i++)
      r[i-1] = p[i] * cdouble(i);
    return r;
  }

  pair<poly, cdouble> ruffini(cdouble z) {
    if (n == 0) return make_pair(poly(), 0);
    poly r(n-1);
    for (int i = n; i > 0; i--) r[i-1] = r[i] * z + p[i];
    return make_pair(r, r[0] * z + p[0]);
  }

  cdouble operator ()(cdouble z) { return ruffini(z).second; }
```

```
  cdouble find_one_root(cdouble x) {
    poly p0 = *this, p1 = ~p0, p2 = ~p1;
    int m = 10000; //change that for efficiency / precision
    while (m--) {
      cdouble y0 = p0(x); if (cmp(y0) == 0) break;
      cdouble G = p1(x) / y0;  cdouble H = G * G - p2(x) - y0;
      cdouble R = sqrt(cdouble(n-1) * (H * cdouble(n) - G * G));
      cdouble D1 = G + R, D2 = G - R;
      cdouble a = cdouble(n) / (cmp(D1, D2) > 0 ? D1 : D2);
      x -= a; if (cmp(a) == 0) break;
    }
    return x;
  }

  vector<cdouble> roots() {
    poly q = *this; vector<cdouble> r;
    while (q.n > 1) {
      cdouble z(rand()/double(RAND_MAX), rand()/double(RAND_MAX));
      z = q.find_one_root(z); z = find_one_root(z);
      q = q.ruffini(z).first;
      r.push_back(z);
    }
    r.push_back(-q[0]/q[1]);
    return r;
  }
};

/*************************************************************************
* Longest Increasing Subsequence in O(n log n)                           *
*************************************************************************/
#define bs(x,v) (upper_bound((x).begin(),(x).end(),v)-(x).begin()-1)
long longest_increasing_subsequence(vector<long> &x)
{
  int l=0; // longest increasing subsequence length
  vector<long> m; //position ofthe lowest element on LIS of length i
  vector<long> xm; // maintain the x[m[i]]
  vector<long> p(x.size(), -1);// maintain the father
  rep(i,x.size()) {
    int j = bs(xm,x[i]); if (j == -1) j=0;
    if (j == xm.size()) { m.push_back(-1); xm.push_back(INF); }
    p[i]=m[j];
    if (j == l || x[i] < xm[j+1]) {
      if (j+1 == xm.size()) { xm.push_back(INF); m.push_back(-1); }
      m[j+1] = i; xm[j+1] = x[i]; l = max(l, j+1);
    }
  }
  return l;
}

/*************************************************************************
* Room Assignment Problem ( Generalized Activity Selection )             *
*************************************************************************/
typedef pair<int,int> activity; //(end,begin)
vector<activity> possible;
```

```
int assign(int rooms) {
  sort( possible.begin(), possible.end() );
  int N=0, last[rooms]; rep(i,rooms) last[i]=-1;

  rep(i,possible.size()) {
    for(int k=rooms-1; k>=0; --k) {
      sort(last, last+rooms);
      if ( last[k] <= possible[i].second ) {
        last[k] = possible[i].first; ++N; break;
      }
    }
  }
  return N;
}

/**************************************************************************
* Knapsack Problem: max Sum(v[i] x[i]) s.t. Sum(w[i] x[i]) <= W           *
**************************************************************************/
int knapsack( vector<int> &v, vector<int> &w, int W) {
  int A[W+1]; int m = w.size(); A[0] = 0;
  for (int i=1; i<=W; i++){
    A[i] = A[i-1];
    rep(j,m) if (w[j] < i) A[i] >?= A[i-w[j]] + v[j];
  }
  return A[W];
}

/**************************************************************************
* 0-1 Knapsack Problem                                                    *
**************************************************************************/
int knapsack01 (vector<int> v, vector<int> w, int W) {
  int m = v.size(); int A[m+1][W+1];
  for (int i=0; i<=m; i++) A[i][0] = 0;
  for (int j=0; j<=W; j++) A[0][j] = 0;
  for (int i=1; i<=m; i++)
    for (int j=1; j<=W; j++) {
      if (w[i-1] > j)
        A[i][j] = A[i-1][j];
      else
        A[i][j] = max(A[i-1][j], v[i-1] + A[i-1][j-w[i-1]]);
    }
  return A[m][W];
}

/**************************************************************************
* Levenshtein Distance                                                    *
**************************************************************************/
int levenshteinDistance (string a, string b) {
  int cost, insertionCost=1, deletionCost=1, substitutionCost=1;
  int m = a.length(); int n = b.length(); int d[m+1][n+1];
  for (int i=0; i<=m; i++) d[i][0] = i;
  for (int j=0; j<=n; j++) d[0][j] = j;
  for (int i=1; i<=m; i++)
    for (int j=1; j<=n; j++) {
      if (a[i-1]==b[j-1]) cost = 0;
```

```
        else cost = substitutionCost;

      d[i][j] = min( d[i-1][j] + deletionCost,
             min( d[i][j-1] + insertionCost, d[i-1][j-1] + cost ));
    }
    return d[m][n];
}


/**************************************************************************
* Geometry                                                                *
**************************************************************************/
struct point {
   double x, y;
   point(double x = 0, double y = 0): x(x), y(y) {}
   point operator +(point q) { return point(x + q.x, y + q.y); }
   point operator -(point q) { return point(x - q.x, y - q.y); }
   point operator *(double t) { return point(x * t, y * t); }
   point operator /(double t) { return point(x / t, y / t); }
   double operator *(point q) { return x * q.x + y * q.y; }
   double operator %(point q) { return x * q.y - y * q.x; }
   int cmp(point q) const {
     if (int t = ::cmp(x, q.x)) return t;
     return ::cmp(y, q.y);
   }
   bool operator ==(point q) const { return cmp(q) == 0; }
   bool operator !=(point q) const { return cmp(q) != 0; }
   bool operator < (point q) const { return cmp(q) < 0; }
   friend ostream& operator <<(ostream& o, point p) {
     return o << "(" << p.x << ", " << p.y << ")";
   }
   static point pivot;
};

point point::pivot;

double abs(point p) { return hypot(p.x, p.y); }
double arg(point p) { return atan2(p.y, p.x); }

typedef vector<point> polygon;

int ccw(point p, point q, point r) {
   return cmp((p - r) % (q - r));
}
double angle(point p, point q, point r) {
   point u = p - q, v = r - q;
   return atan2(u % v, u * v);
}

/**************************************************************************
* Decides if q is on the closed segment [pr]                              *
**************************************************************************/
bool between(point p, point q, point r) {
   return ccw(p, q, r) == 0 && cmp((p - q) * (r - q)) <= 0;
}
```

```
/*****************************************************************
* Decides if the closed segments [pq] and [rs] have common points   *
*****************************************************************/
bool seg_intersect(point p, point q, point r, point s) {
   point A = q - p, B = s - r, C = r - p, D = s - q;
   int a = cmp(A % C) + 2 * cmp(A % D);
   int b = cmp(B % C) + 2 * cmp(B % D);
   if (a == 3 || a == -3 || b == 3 || b == -3) return false;
   if (a || b || p == r || p == s || q == r || q == s) return true;
   int t = (p < r) + (p < s) + (q < r) + (q < s);
   return t != 0 && t != 4;
}


/*****************************************************************
* Distance between point r and segment [pq]                        *
*****************************************************************/
double seg_distance(point p, point q, point r) {
   point A = r - q, B = r - p, C = q - p;
   double a = A * A, b = B * B, c = C * C;
   if (cmp(b, a + c) >= 0) return sqrt(a);
   else if (cmp(a, b + c) >= 0) return sqrt(b);
   else return fabs(A % B) / sqrt(c);
}

/*****************************************************************
* Returns 0 ( exterior ), -1 ( border ), 1 ( interior )            *
*****************************************************************/
int in_poly(point p, polygon& T) {
   double a = 0; int N = T.size();
   for (int i = 0; i < N; i++) {
     if (between(T[i], p, T[(i+1) % N])) return -1;
     a += angle(T[i], p, T[(i+1) % N]);
   }
   return cmp(a) != 0;
}

/*****************************************************************
* Convex Hull ( WARNING: destroys vector T )                       *
*****************************************************************/
bool radial_lt(point p, point q) { // radial comparison
   point P = p - point::pivot, Q = q - point::pivot;
   double R = P % Q;
   if (cmp(R)) return R > 0;
   return cmp(P * P, Q * Q) < 0;
}


polygon convex_hull(vector<point>& T) {
   int j = 0, k, n = T.size(); polygon U(n);
   point::pivot = *min_element(all(T));
   sort(all(T), radial_lt);
   for (k = n-2; k >= 0 && ccw(T[0], T[n-1], T[k]) == 0; k--);
   reverse((k+1) + all(T));
```

```
  for (int i = 0; i < n; i++) {
    // substitute >= for > to mantain collinear points
    while (j > 1 && ccw(U[j-1], U[j-2], T[i]) >= 0) j--;
    U[j++] = T[i];
  }
  U.erase(j + all(U));
  return U;
}

/*******************************************************************
* Calculates ordered area from polygon T                           *
*******************************************************************/
double poly_area(polygon& T) {
  double s = 0; int n = T.size();
  for (int i = 0; i < n; i++)
    s += T[i] % T[(i+1) % n];
  return s / 2;
}

/*******************************************************************
* Finds intersection point from lines pq and rs                    *
*******************************************************************/
point line_intersect(point p, point q, point r, point s) {
  point a = q - p, b = s - r, c = point(p % q, r % s);
  return point(point(a.x, b.x) % c, point(a.y, b.y) % c) / (a % b);
}

/*******************************************************************
* Spanning circle                                                  *
*******************************************************************/
typedef pair<point, double> circle;

bool in_circle(circle C, point p){
  return cmp(abs(p - C.first), C.second) <= 0;
}

point circumcenter(point p, point q, point r) {
  point a = p-r, b = q-r, c = point(a*(p+r)/2, b*(q+r)/2);
  return point(c%point(a.y, b.y), point(a.x, b.x)%c) / (a%b);
}

circle spanning_circle(vector<point>& T) {
  int n = T.size();
  random_shuffle(all(T));
  circle C(point(), -INFINITY);
  for (int i = 0; i < n; i++) if (!in_circle(C, T[i])) {
    C = circle(T[i], 0);
    for (int j = 0; j < i; j++) if (!in_circle(C, T[j])) {
      C = circle((T[i] + T[j]) / 2, abs(T[i] - T[j]) / 2);
      for (int k = 0; k < j; k++) if (!in_circle(C, T[k])) {
        point o = circumcenter(T[i], T[j], T[k]);
        C = circle(o, abs(o - T[k]));
      }
    }
  }
```

```
    return C;
}

/*******************************************************************
* Polygon Intersection (both polygons must be positively oriented)  *
*******************************************************************/
polygon poly_intersect(polygon& P, polygon& Q) {
    int m = Q.size(), n = P.size();
    int a = 0, b = 0, aa = 0, ba = 0, inflag = 0;
    polygon R;
    while ((aa < n || ba < m) && aa < 2*n && ba < 2*m) {
      point p1=P[a], p2=P[(a+1)%n], q1=Q[b], q2=Q[(b+1)%m];
      point A = p2 - p1, B = q2 - q1;
      int cross=cmp(A % B), ha=ccw(p2, q2, p1), hb=ccw(q2, p2, q1);
      if (cross == 0 && ccw(p1, q1, p2) == 0 && cmp(A * B) < 0) {
        if (between(p1, q1, p2)) R.push_back(q1);
        if (between(p1, q2, p2)) R.push_back(q2);
        if (between(q1, p1, q2)) R.push_back(p1);
        if (between(q1, p2, q2)) R.push_back(p2);
        if (R.size() < 2) return polygon();
        inflag = 1; break;
      } else if (cross != 0 && seg_intersect(p1, p2, q1, q2)) {
        if (inflag == 0) aa = ba = 0;
        R.push_back(line_intersect(p1, p2, q1, q2));
        inflag = (hb > 0) ? 1 : -1;
      }
      if (cross == 0 && hb < 0 && ha < 0) return R;
      bool t = cross == 0 && hb == 0 && ha == 0;
      if (t ? (inflag==1) : (cross>=0) ? (ha<=0) : (hb>0)) {
        if (inflag == -1) R.push_back(q2);
        ba++; b++; b %= m;
      } else {
        if (inflag == 1) R.push_back(p2);
        aa++; a++; a %= n;
      }
    }
    if (inflag == 0) {
      if (in_poly(P[0], Q)) return P;
      if (in_poly(Q[0], P)) return Q;
    }
    R.erase(unique(all(R)), R.end());
    if (R.size() > 1 && R.front() == R.back()) R.pop_back();
    return R;
}

/*******************************************************************
* Voronoi Diagram                                                  *
*******************************************************************/
int B; point sites[60];

point orthogonal( point p1, point p2 ) {
    point v =  p2 - p1; point dir( v.y, -v.x ); dir = dir / abs(dir);
    return dir;
}
```

```
// Halfplane containing the points closer to ti than to tj
polygon half_plane( point ti, point tj ) {
  polygon R;
  point mid = ( ti + tj )*.5;
  point dir = orthogonal( ti, tj );

  point p1 = mid + dir * INF; point p2 = mid - dir * INF;
  point p3 = p2 + ( ti - tj ) * INF;
  point p4 = p1 + ( ti - tj ) * INF;;

  R.push_back( p1 ); R.push_back( p2 );
  R.push_back( p3 ); R.push_back( p4 );
  return R;
}

void voronoi( vector<polygon>& cells )
{
  cells.resize(B);
  rep(i,B) {
    polygon pi;
    pi.push_back( point(-INF, -INF ) );
    pi.push_back( point( INF, -INF ) );
    pi.push_back( point( INF,  INF ) );
    pi.push_back( point(-INF,  INF ) );
    rep(j,B) if ( i != j ) {
      polygon h = half_plane( sites[i], sites[j] );
      pi = poly_intersect( pi, h );
    }
    rep(j,pi.size()) cells[i].push_back( pi[j] );
  }
}

/*****************************************************************
* Bresenham Algorithm - draws a line in a grid                   *
*****************************************************************/
void plot(int x, int y) {}//do something

void bresenham(int x1, int y1, int x2, int y2) {
  int delta_x = std::abs(x2 - x1) << 1;
  int delta_y = std::abs(y2 - y1) << 1;
  signed char ix = x2 > x1?1:-1;
  signed char iy = y2 > y1?1:-1;
  plot(x1, y1);

  if (delta_x >= delta_y) { // error may go below zero
    int error = delta_y - (delta_x >> 1);
    while (x1 != x2) {
      if (error >= 0)
         if (error || (ix > 0)) { y1 += iy; error -= delta_x; }
      x1 += ix;
      error += delta_y;
      plot(x1, y1);
    }
  } else {
    int error = delta_x - (delta_y >> 1);
```

```
    while (y1 != y2) {
      if (error >= 0)
        if (error || (iy > 0)) { x1 += ix; error -= delta_y; }
      y1 += iy;
      error += delta_x;
      plot(x1, y1);
    }
  }
}
```

```
/*****************************************************************
* Maximum Bipartite Matching                                     *
*****************************************************************/
#define WHITE 0
#define GRAY 1
int N,M; //sizes of the S and T sets
vector<int> adj[NMAX]; //adj[u][i]=v -> u in (0...n-1); v in (0..m-1)
int color[NMAX];
vector<int> path; //path contains only right side vertices
int match[MMAX]; //-1 or the vertex in S {0..n-1} matched

int aug(int u) {
  if (color[u] == WHITE) {
    color[u] = GRAY;
    rep(i,adj[u].size()) {
      int v = adj[u][i]; int w = match[v]; path.push_back(v);
      if (w == -1) return 1;
      else {
        if (aug(w)) return 1;
        else path.pop_back();
      }
    }
  }
  return 0;
}

int matching() {
  int u, v, w, flow = 0;
  rep(i,M) match[i] = -1;
  rep(i,N) {
    //Solve matching
    rep(j,N) color[j] = WHITE;
    path.clear();
    flow += aug(i);

    //Update path
    u = i;
    rep(j,path.size()) {
      w = path[j]; v = match[w]; match[w] = u; u = v;
    }
  }
  return flow;
}

/*****************************************************************
* Flow Definitions                                               *
*****************************************************************/
#define inv(e) ((e)^0x1)
#define origin(e) dest[ inv(e) ]
#define res_cap(e) upper[e] - flow[e]

vector<int> adj[NMAX]; int dest [ 2 * MMAX ]; int flow [ 2 * MMAX ];
int cost [ 2 * MMAX ]; int upper[ 2 * MMAX ]; //upper bound capacity
int pi[NMAX]; int pot[NMAX]; int d[NMAX]; int n,m;
//Initialize with n = ...; m = 0; rep(i,n) adj[i].clear();
```

```
void add_edge( int a, int b, int up, int co ) {
  dest[m] = b; upper[m] = up; cost[m] = co;
  adj[a].push_back(m++);
  dest[m] = a; upper[m] = 0; cost[m] = -co;
  adj[b].push_back(m++);
}


/**********************************************************************
* Min Cost Max Flow                                                   *
**********************************************************************/
bool bellman_ford(int s) {
  rep(i,n) pot[i] = INF; pot[s] = 0;
  rep(k,n) rep(u,n) rep(i,adj[u].size()) if (res_cap(adj[u][i]) > 0)
    pot[ dest[ adj[u][i] ] ] <?= pot[ u ] + cost[ adj[u][i] ];
  rep(u,n) rep(i,adj[u].size())
    if ( ( res_cap( adj[u][i] ) > 0 ) &&
        (pot[ dest[ adj[u][i] ] ] > pot[ u ] + cost[ adj[u][i] ]) )
      return false;
  return true; //no negative cycle
}

int dijkstra( int s, int t ) {
  ii top; int u,e,v,c, dist;
  rep(i,n) { d[i] = INF; pi[i] = -1; }
  set<ii> Q; Q.insert( ii(0,s) ); d[s] = 0;

  while (!Q.empty()) {
    ii top = *Q.begin(); Q.erase(Q.begin());
    u = top.second, dist = top.first;
    rep(i, adj[u].size()) if ( res_cap( adj[u][i] ) > 0 ) {
      e = adj[u][i]; v = dest[e]; c = pot[u] + cost[e] - pot[v];
      if ( d[v] > d[u] + c ) {
        if ( d[v]!=INF ) Q.erase( Q.find( ii(d[v],v) ) );
        d[v] = d[u] + c; pi[v] = e; Q.insert( ii(d[v],v) );
      }
    }
  }
  rep(i,n) if (d[i]!=INF) pot[i] += d[i];
  return d[t];
}

void min_cost_max_flow(int s, int t, int &fflow, int &fcost)
{
  int u, pflow; fflow = fcost = 0; rep(i,m) flow[i] = 0;
  rep(i,n) pot[i] = 0; // or bellman_ford(s);

  while ( 1 ) {
    dijkstra(s,t);
    if ( pi[t] != -1 ) {
      pflow = INF;
      for (u = t; u!=s; u = origin(pi[u])) {
        pflow <?= res_cap( pi[u] );
      }
```

```
      for (u = t; u!=s; u = origin(pi[u])) {
        flow[ pi[u] ] += pflow; flow[ inv(pi[u]) ] -= pflow;
        fcost += pflow*cost[ pi[u] ];
      }
      fflow += pflow;
    }
    else return;
  }
}

/**************************************************************************
* Max Flow                                                                *
**************************************************************************/
int max_flow(int s, int t)
{
  int u, v, e, pflow, ans = 0; rep(i,m) flow[i] = 0;

  vector<int> path;
  path.reserve(n);

  while(1) {
    //augment path
    rep(i,n) pi[i] = -2;
    queue<int> Q; Q.push(s); pi[s] = -1;
    while (!Q.empty()) {
      u = Q.front(); Q.pop();
      rep(i,adj[u].size()) {
        e = adj[u][i]; v = dest[e];
        if ( pi[v] == -2 && res_cap(e) > 0 ) {
          pi[v] = e; Q.push(v); if ( v == t ) break;
        }
      }
    }

    if ( pi[t] != -2 ) {
      pflow = INF;
      for (u = t; u!=s; u=origin(pi[u])) pflow<?=res_cap( pi[u] );
      for (u = t; u!=s; u = origin(pi[u])) {
        flow[ pi[u] ] += pflow; flow[ inv(pi[u]) ] -= pflow;
      }
      ans += pflow;
    }
    else return ans;
  }
}

/**************************************************************************
* 2-Satisfiability and Strongly Connected Components                      *
* N is the number of variables and n the number of vertices. We           *
* have n = 2*N ( one vertex for x_i and other for ~x_i ).                 *
* x_i OR x_j <--> " ~x_j -> x_i AND ~x_i -> x_j "                         *
* For example to add the clausule u OR ~v, we do:                         *
* adj[v].push_back(u); adj[NOT(u)].push_back(NOT(v));                     *
* Don't forget rep(i,n) adj[i].clear();                                   *
**************************************************************************/
```

```
int N; int n; const int NMAX = 1000; vector<int> adj[2*NMAX];
int ncomp; int comp[2*NMAX]; // strongly connected component
#define NOT(i) ((i<N) ? (i+N) : (i-N))

vector<int> adjT[2*NMAX]; int seen[2*NMAX]; int order[2*NMAX];
int norder;

void dfs1(int u) {
  seen[u] = 1;
  rep(i,adj[u].size()) if (!seen[adj[u][i]]) dfs1( adj[u][i] );
  order[norder++] = u;
}

void dfs2(int u) {
  seen[u] = 1;
  rep(i,adjT[u].size()) if (!seen[adjT[u][i]]) dfs2( adjT[u][i] );
  comp[u] = ncomp;
}

void strongly_connected_components() {
  rep(u,n) adjT[u].clear();
  rep(u,n) rep(j,adj[u].size()) adjT[ adj[u][j] ].push_back(u);

  norder = 0; rep(i,n) seen[i] = 0;
  rep(i,n) if (!seen[i]) dfs1(i);

  rep(i,n) seen[i] = 0; ncomp = 0;
  for(int i=n-1, u = order[n-1]; i>=0; u = order[--i])
    if (!seen[u]) { dfs2(u); ncomp++;}
}

bool sat2() {
  strongly_connected_components();
  rep(i,N) if ( comp[i] == comp[ NOT(i) ] ) return false;
  return true;
}

/*************************************************************************
* If sat2() return 1, we can use sat2_solution to get an explicit        *
* solution of the problem. It is stored in x[NMAX]                       *
*************************************************************************/
int x[2*NMAX];
int dfs3( int u ) {
  x[u] = 1; x[ NOT(u) ] = 0;
  rep(i,adj[u].size()) if (x[adj[u][i]]==-1) dfs3( adj[u][i] );
}

void sat2_solution()
{
  rep(i,n) x[i] = -1;
  rep(i,n) if (x[i] == -1)
    if ( comp[i] > comp[ NOT(i) ] ) dfs3(i);
    else dfs3( NOT(i) );
}
```

```
/*****************************************************************
* Steiner Tree - use: n is the number os vertices, the graph is in  *
* d[n][n], active_vertex[n] is a bitmap indicating which vertices   *
* should be connected, size is the number of vertices to be         *
* connected and their indexes are stored in C[0...size-1]. To solve *
* the problem, you need to call floyd_warshall() and then           *
* steiner_tree(size-2,0);                                           *
*****************************************************************/
int n, size; bool active_vertex[30]; int C[30];

int d[30][30]; int gpi[30][30];

void floyd_warshall()
{
  rep(i,n) d[i][i] = 0;
  rep(i,n) rep(j,n) gpi[i][j] = i;
  rep(k,n) rep(i,n) rep(j,n) {
    if  ( d[i][j] > d[i][k] + d[k][j] ) {
      d[i][j] = d[i][k] + d[k][j];
      gpi[i][j] = gpi[k][j];
    }
  }
}

int D[30]; bool visited[30]; int pi[30];

int mst()
{
  int u, cost, res = 0;
  rep(i,size) { D[i] = INF; visited[i] = false; }
  D[0] = 0; pi[ C[0] ] = -1;

  rep(k,size) {
    u = -1; cost = INF;
    rep(i,size) if ( (!visited[i]) && ( cost > D[i]) ) {
      u = i; cost = D[i];
    }

    res += cost; visited[u] = true;
    rep(i,size) if ( !visited[i] ) {
      if ( D[i] > d[ C[u] ][ C[i] ] ) {
        D[i] = d[ C[u] ][ C[i] ]; pi[ C[i] ] = C[u] ;
      }
    }
  }
  // do something with the answer here
  return res;
}


int steiner_tree(int internal, int from) {
  if (internal <= 0) return mst();
  int res = mst();
  for(int i=from; i<n; ++i) {
```

```
    if (!active_vertex[i]) {
      active_vertex[i] = true; C[size++] = i;
      res <?= steiner_tree(internal-1, i+1);
      --size; active_vertex[i] = false;
    }
  }
  return res;
}

/*************************************************************************
* Articulations and Bridges: first verify if the graph is connected *
*************************************************************************/
typedef pair<int, int> edge;
int n; vector<int> adj[NMAX];
vector<int> articulation; vector<edge> bridge;

bool visited[NMAX]; int d[NMAX], lowpt[NMAX], pi[NMAX]; int timer;

void dfs(int u) {
  int sons=0, v; bool art = false;
  visited[u] = true; lowpt[u] = d[u] = timer++;

  rep(i,adj[u].size()) {
    v = adj[u][i];
    if (!visited[ adj[u][i] ]) {
      sons++; pi[v] = u; dfs(v);
      lowpt[u] <?= lowpt[v];
      if ((pi[u]!=-1) && (lowpt[v] >= d[u])) art = true;
      if (lowpt[v] > d[u]) bridge.push_back(edge(u,v));
    }
    else if ( v!= pi[u] ) lowpt[u] <?= d[v];
  }

  if (art) articulation.push_back(u);
  else if (pi[u] == -1 && sons > 1 ) articulation.push_back(u);
}

void articulations_and_bridges() {
  articulation.clear(); bridge.clear(); timer=0;
  rep(i,n) { visited[i] = false; pi[i] = -1; }
  rep(i,n) if (!visited[i]) dfs(i);
}

/*************************************************************************
* Tree Isomorphism: Both trees should be stored in adj, the first   *
* tree with nodes 0...n-1 and the second with nodes n...2n-1        *
*************************************************************************/
#define all(v) (v).begin(),(v).end()
#define degree(v) adj[v].size()
deque<int> L[1500]; //levels
deque<int> sons[3000]; //levels
deque<int> subtreeLabels[3000];
int pi[3000]; int label[3000]; int color[3000];
int n; deque<int> adj[3000]; int map[3000];
```

```
bool compare(int a, int b){
        return (subtreeLabels[a] < subtreeLabels[b]);
}
bool equals(int a, int b) {
        return (subtreeLabels[a] == subtreeLabels[b]);
}

int DFSVisit(int r, int l = 0){ //l is levels
  int v,h = 0; L[l].push_back(r);
  color[r] = GRAY;
  for(int i=0; i<adj[r].size(); i++) {
    v = adj[r][i];
    if (color[v]==WHITE) {
      pi[v] = r; sons[r].push_back(v);
      h = max(h, DFSVisit(v,l+1));
    }
  }
  color[r] = BLACK;
  return h+1;
}

void GenerateMapping(int r1, int r2) {
  int u,v; map[r1] = r2-n;
  sort(all(sons[r1]),compare); sort(all(sons[r2]),compare);
  for(int i=0; i<sons[r1].size(); i++){
    u = sons[r1][i]; v = sons[r2][i]; GenerateMapping(u,v);
  }
}

bool RootedTreeIsomorphism(int r1, int r2) {
  for(int i=0; i<n; i++) L[i].clear();
  for(int i=0; i<2*n; i++) {
    pi[i] = -1; color[i] = WHITE; sons[i].clear();
    subtreeLabels[i].clear();
  }
  int h1 = DFSVisit(r1);
  int h2 = DFSVisit(r2);
  if (h1!=h2) return false;
  else {
    int h = h1-1;
    for (int i=0; i<L[h].size(); i++) label[L[h][i]] = 0;
    for (int i = h-1; i>=0; i--) {
      for (int j=0; j<L[i+1].size(); j++)
      subtreeLabels[pi[L[i+1][j]]].push_back(label[L[i+1][j]]);

      for (int j=0; j<L[i+1].size(); j++)
      sort(subtreeLabels[L[i+1][j]].begin(),
                subtreeLabels[L[i+1][j]].end());

      sort(L[i].begin(), L[i].end(), compare);
      int actualLabel = 0;
      label[L[i][0]] = actualLabel;

      for (int j=1; j<L[i].size(); j++) {
        if (!equals(L[i][j], L[i][j-1])) actualLabel++;
```

```
        label[L[i][j]] = actualLabel;
      }
    }

    if (equals(r1, r2)) { GenerateMapping(r1,r2); return true; }
    else return false;
  }
}

deque<int> FindCenter(int k) //k=0 is tree 1, k=1 is tree 2
{
  deque<int> S, T;
  int r, u, v, cdeg[n]; r = n;
  for(int i=0; i<n;i++) {
    cdeg[i] = degree(i+k*n);
    if (cdeg[i] <=1 ) { S.push_back(i+k*n); r--; }
  }

  while (r>0) {
    T.clear();
    for(int i=0; i<S.size(); i++) {
      u = S[i];
      for(int j=0; j<adj[u].size(); j++) {
        v = adj[u][j]; cdeg[v-k*n]--;
        if (cdeg[v-k*n] == 1) {
          T.push_back(v); r--;
        }
      }
    }
    S = T;
  }
  return S;
}

bool TreeIsomorphism() {
  deque<int> S1 = FindCenter(0);
  deque<int> S2 = FindCenter(1);
  if (S1.size() != S2.size()) return false;
  else {
    if (RootedTreeIsomorphism(S1[0],S2[0])) return true;
    else if (S1.size()>1) return RootedTreeIsomorphism(S1[0],S2[1]);
    else return false;
  }
}

/*****************************************************************
* Least Common Ancestor <O(n log n), O(log n)> : store the tree     *
* in pi, N the number of vertices, call pre_processing()            *
*****************************************************************/
typedef pair<int,int> ii;
const int NMAX = 10000;
const int LOGNMAX = 14;
int pi[ NMAX ]; //father
int L [ NMAX ]; //level
int P [ NMAX ][ LOGNMAX ]; int N;
```

```
int go_level( int u ) {
  if ( L[u] != -1 ) return L[u];
  else if ( pi[u] == -1 ) return 0;
  else return 1 + go_level ( pi[u] );
}

void pre_processing() {
  rep(i,N) L[i] = -1;
  rep(i,N) if ( L[i] == -1 ) L[i] = go_level(i);
  rep(i,N) for(int j = 0; 1<<j < N; ++j) P[i][j] = -1;
  rep(i,N) P[i][0] = pi[i];
  for(int j=1; 1<<j < N; ++j) rep(i,N)
    if ( P[i][j-1] != -1 )
      P[i][j] = P[ P[i][j-1] ][j-1];
}

int LCA ( int p, int q ) {
  if ( L[p] < L[q] ) swap(p, q);
  int log; for ( log = 1; 1 << log <= L[p]; ++log ); --log;
  for (int i = log; i>=0; --i)
    if ( L[p] - ( 1 << i ) >= L[q] ) p = P[p][i];
  if ( p == q ) return p;
  for (int i = log; i>=0; --i) {
    if ( P[p][i] != -1 && P[p][i] != P[q][i] ) {
      p = P[p][i]; q = P[q][i];
    }
  }
  return pi[p];
}

/*************************************************************************
* Dijkstra with Heap                                                     *
*************************************************************************/
vector< ii > adj[NMAX]; //(u,v) --> adj[u][i] = (v, cost)
int N; int D[NMAX], pi[NMAX];

void dijkstra(int s) { //fills D and pi
  rep(i,N) { D[i] = INF; pi[i] = -1; }
  priority_queue<ii,vector<ii>, greater<ii> > Q;
  D[s] = 0; Q.push(ii(0,s));

  while(!Q.empty()) {
    ii top = Q.top(); Q.pop();
    int u = top.second, d = top.first;
    if (d<=D[u]) rep(i,adj[u].size()) {
      int v = adj[u][i].first, cost = adj[u][i].second;
      if ( D[v] > D[u] + cost ) {
        D[v] = D[u] + cost;
        pi[v] = u;
        Q.push( ii( D[v], v ) );
      }
    }
  }
}
```

```cpp
/*************************************************************************
* Prim with heap                                                         *
*************************************************************************/
vector< ii > adj[NMAX]; //(u,v) --> adj[u][i] = (v, cost)
int N; int D[NMAX], pi[NMAX]; bool visited[NMAX];

int prim() //fills D and pi and returns cost of the mst
{
  int ans = 0;
  rep(i,N) { D[i] = INF; pi[i] = -1; visited[i] = false; }
  priority_queue<ii,vector<ii>, greater<ii> > Q;
  D[0] = 0; Q.push(ii(0,0));

  while(!Q.empty()) {
    ii top = Q.top(); Q.pop();
    int u = top.second, d = top.first;

    if (!visited[u])
    {
      ans += d; visited[u] = true;
      rep(i,adj[u].size()) {
        int v = adj[u][i].first, cost = adj[u][i].second;
        if ( !visited[v] && ( D[v] > cost ) ) {
          D[v] = cost; pi[v] = u;
          Q.push( ii( D[v], v ) );
        }
      }
    }
  }
  return ans;
}
```