# Universidade Federal de Goiás
# Team Reference Material

2015 South America/Brazil Regional Contest

## Contents

# Template

```
#include <bits/stdc++.h>

#define FILL(X, V) memset((X), (V), sizeof(X))
#define TI(X) __typeof((X).begin())
#define ALL(V) V.begin(), V.end()
#define SIZE(V) int((V).size())

#define FOR(i, a, b) for(int i = a; i <= b; ++i)
#define RFOR(i, b, a) for(int i = b; i >= a; --i)
#define REP(i, N) for(int i = 0; i < N; ++i)
#define RREP(i, N) for(int i = N-1; i >= 0; --i)
#define FORIT(i, a) for(TI(a) i = (a).begin(); i != (a).end(); ++i)

#define pb push_back
#define mp make_pair

#define INF 0x3F3F3F3F
#define LINF 0x3F3F3F3FFFFFFFFFLL

const double EPS = 1e-9;

int SGN(double a){ return ((a > EPS) ? (1) : ((a < -EPS) ? (-1) : (0))); }
int CMP(double a, double b){ return SGN(a - b); }

typedef long long int64;
typedef unsigned long long uint64;
typdef pair<int, int> ii;

struct node{
   int a, b;
   node (int a = 0, int b = 0) : a(a), b(b) {}
};

using namespace std;

int main(int argc, char* argv[]){
   ios::sync_with_stdio(false);

   return 0;
}
```

# Combinatorics

## Binomial Coefficients

Number of ways to pick a multiset of sike $k$ from $n$ elements: $\binom{n+k-1}{k}$
Number of $n$-tuples of non-negative integers with sum $s$: $\binom{s+n-1}{n-1}$, at most $s$: $\binom{s+n}{n}$
Number of $n$-tuples of positive integers with sum $s$: $\binom{s-1}{n-1}$
Number of lattice paths from $(0,0)$ to $(a,b)$, restricted to east and north steps: $\binom{a+b}{a}$

## Pascal Triangle

$v_{r,c} = v_{r,c-1}\frac{r+1-c}{c}$
$x = r_0$; $y = 1$; $a_{r,c} = a_{r-1,c-1}\frac{++x}{++y}, r \geq r_0 + 2, c \geq 2$

Line on a Pascal Triangle: $c(n,k+1) = c(n,k) \times \frac{n-k}{k+1}$

## Log properties

$log(a^n) = n \times log(a)$
$log(n!) = \sum_{i=1}^{n} log(i)$

## Catalan numbers

$C_n = \frac{1}{n+1}\binom{2n}{n}$. $C_0 = 1$, $C_n = \sum_{i=0}^{n-1} C_i C_{n-1-i}$. $C_{n+1} = C_n \frac{4n+2}{n+2}$.
$C_0, C_1, ... =$ 1, 1, 2, 5, 14, 42, 132, 429, 1430, 4862, 16796, 58786, 208012, 742900, ...
$C_n$ is the number of: properly nested sequences of $n$ pais of parentheses; rooted ordered binary trees with $n+1$ leaves; triangulations of a convex $(n+2)$-gon.

## Derangements

Number of permutations of $n = 0, 1, 2, ...$ elements without fixed points is 1, 0, 1, 2, 9, 44, 265, 1854, 14833, ... Recurrence: $D_n = (n-1)(D_{n-1} + D_{n-2}) = nD_{n-1} + (-1)^n$. Corollary: number of permutations with exactly $k$ fixed points is $\binom{n}{k}D_{n-k}$.

## Bell numbers

$B_n$ is the number of partitions of $n$ elements. $B_0$, ... = 1, 1, 2, 5, 15, 52, 203, ...
$B_{n+1} = \sum_{k=0}^{n}\binom{n}{k}B_k = \sum_{k=1}^{n} S_{n,k}$. Bell triangle: $B_r = a_{r,1} = a_{r-1,r-1}, a_{r,c} = a_{r-1,c-1} + a_{r,c-1}$.

### Eulerian numbers

$E(n, k)$ is the number of permutations with exactly $k$ descents $(i : \pi_i < \pi_{i+1})$ / ascents $(\pi_i > \pi_{i+1})$ / excedances $(\pi_i > i)$ / $k+1$ weak excedances $(\pi_i \geq i)$.
Formula: $E(n, k) = (k+1)E(n-1, k) + (n-k)E(n-1, k-1)$.

### Burnside's lemma

The number of orbits under $G$'s action on set $X$ : $|X/G| = \frac{1}{|G|}\sum_{g\in G}|X_g|$, where $X_g = \{x \in X : g(x) = x\}$. ("Average number of fixed points.") Let $w(x)$ be weight of $x$'s orbit. Sum of all orbits' weights: $\sum_{o\in X/G} w(o) = \frac{1}{|G|}\sum_{g\in G}\sum_{x\in X_g} w(x)$.

## Number Theory

### Linear diophantine equation

$ax + by = c$. Let $d = gcd(a, b)$. A solution exists iff $d|c$. If $(x_0, y_0)$ is any solution, then all solutions are given by $(x, y) = (x_0 + \frac{b}{d}t, y_0 - \frac{a}{d}t), t \in \mathbb{Z}$. To find some solution $(x_0, y_0)$, use extended GCD to solve $ax_0 + by_0 = d = gcd(a, b)$, and multiply its solutions by $\frac{c}{d}$.
Linear diophantine equation in $n$ variables: $a_1x_1 + ... + a_nx_n = c$ has solutions iff $gcd(a_1, ..., a_n)|c$. To find some solution, let $b = gcd(a_2, ..., a_n)$, solve $a_1x_1 + by = c$, and iterate with $a_2x_2 + ... = y$.

### Chinese Remainder Theorem

System $x \equiv a_i(mod\ m_i)$ for $i = 1, ..., n$ with pairwise relatively prime $m_i$ has a unique solution modulo $M = m_1m_2...m_n : x = a_1b_1\frac{M}{m_1} + ... + a_nb_n\frac{M}{m_n}(mod\ M)$, where $b_i$ is modular inverse of $\frac{M}{M_i}$ modulo $m_i$.
System $x \equiv a(mod\ m), x \equiv b(mod\ n)$ has solutions iff $a \equiv b(mod\ g)$, where $g = gcd(m, n)$. The solution is unique modulo $L = \frac{mn}{g}$, and equals: $x \equiv a + T(b-a)m/g \equiv b + S(a-b)n/g(mod\ L)$, where $S$ and $T$ are integer solutions of $mT + nS = gcd(m, n)$.

## Extended GCD

```
//return x, y such a * x + b * y = gcd(a, b)
pair<int, int> gcd_extended(int a, int b){
   /*Use only if negative numbers are used as parameters
   if (a < 0){
      pair<int, int> p = gcd_extended(-a, b);
      p.first = -p.first;
      return p;
   }
   if (b < 0){
      pair<int, int> p = gcd_extended(a, -b);
      p.second = -p.second;
      return p;
   }
   */

   int x = 1, y = 0;
   int nx = 0, ny = 1;

   while (b){
      int q = a / b;
      x -= q * nx; swap(x, nx);
      y -= q * ny; swap(y, ny);
      a -= q * b; swap(a, b);
   }

   return mp(x, y);
}

//Reurn a inverse mod b
//gcd(a, b) must be 1
int mod_inv(int a, int b){
   return (gcd_extended(a, b).first + b) % b;
}
```

$$a^{-1}(mod\ n) = a^{\phi(n)-1}$$
$$n\ prime \rightarrow a^{-1}(mod\ n) = a^{n-2}$$

## Prime-counting function

$\pi(n) = |\{p \leq n : p\ is\ prime\}|$. $n/ln(n) < \pi(n) < 1.3n/ln(n)$. $\pi(1000) = 168, \pi(10^6) = 78498, \pi(10^9) = 50847534$. $n$-th prime $\approx nln(n)$.

## Fast Sieve

```
const unsigned MAX = 1000000020/60, MAX_S = sqrt(MAX/60);

unsigned w[16] = {1, 7, 11, 13, 17, 19, 23, 29, 31, 37, 41, 43, 47, 49, 53, 59};
unsigned short composite[MAX];
vector<int> primes;

void sieve(){
   unsigned mod[16][16], di[16][16], num;
   for (int i = 0; i < 16; i++)
      for (int j = 0; j < 16; j++){
         di[i][j] = (w[i]*w[j])/60;
         mod[i][j] = lower_bound(w, w + 16, (w[i]*w[j])%60) - w;
      }

   primes.push_back(2); primes.push_back(3); primes.push_back(5);

   memset(composite, 0, sizeof composite);
   for (unsigned i = 0; i < MAX; i++)
      for (int j = (i==0); j < 16; j++){
         if (composite[i] & (1<<j)) continue;
         primes.push_back(num = 60*i + w[j]);

         if (i > MAX_S) continue;
         for (unsigned k = i, done = false; !done; k++)
            for (int l = (k==0); l < 16 && !done; l++){
               unsigned mult = k*num + i*w[l] + di[j][l];
               if (mult >= MAX) done = true;
               else composite[mult] |= 1<<mod[j][l];
            }
      }
}
```

## Miller-Rabin's primality test

```
int fastpow(int base, int d, int n){
   int ret = 1;
   for (int64 pow = base; d > 0; d >>= 1, pow = (pow * pow) % n)
      if (d & 1)
         ret = (ret * pow) % n;
   return ret;
}

bool miller_rabin(int n, int base){
   if (n <= 1) return false;
   if (n % 2 == 0) return n == 2;

   int s = 0, d = n - 1;
   while (d % 2 == 0) d /= 2, ++s;

   int base_d = fastpow(base, d, n);
   if (base_d == 1) return true;
   int base_2r = base_d;

   for (int i = 0; i < s; ++i){
      if (base_2r == 1) return false;
      if (base_2r == n - 1) return true;
      base_2r = (int64)base_2r * base_2r % n;
   }

   return false;
}

bool isprime(int n){
   if (n == 2 || n == 7 || n == 61) return true;
   return miller_rabin(n, 2) && miller_rabin(n, 7) && miller_rabin(n, 61);
}
```

Given $n = 2^r s + 1$ with odd s, and a random integer $1 < a < n$. If $a^s \equiv 1 (mod\ n)$ or $a^{2^j s} \equiv -1 (mod\ n)$ for some $0 \leq j \leq r - 1$, then $n$ is a probable prime.

## Pollard-$\rho$

```
int64 pollard_r, pollard_n;

int64 f(int64 val){ return (val*val + pollard_r) % pollard_n; }
int64 myabs(int64 a){ return a >= 0 ? a : -a; }

int64 pollard(int64 n){
   srand(unsigned(time(0)));
   pollard_n = n;

   int64 d = 1;
   do{
      d = 1;
      pollard_r = rand() % n;

      int64 x = 2, y = 2;
      while (d == 1)
         x = f(x), y = f(f(y)), d = __gcd(myabs(x-y), n);
   } while (d == n);

   return d;
}
```

Choose random $x_1$, and let $x_{i+1} = x_i^2 (mod\ n)$. Test $gcd(n, x_{2^k+i} - x_{2^k})$ as possible $n$'s factors for $k = 0, 1, ...$ Expected time to find a factor: $O(\sqrt{m})$, where $m$ is smallest prime power in $n$'s factorization. That's $O(n^{1/4})$ if you check $n = p^k$ as a special case before factorization.

## Fermat primes

A Fermat prime is a prime of form $2^{2^n} + 1$. The only known Fermat primes are 3, 5, 17, 257, 65537. A number of form $2^n + 1$ is prime only if it is a Fermat prime.

## Perfect numbers

$n > 1$ is called perfect if it equals sum of its proper divisors and 1. Even $n$ is perfect iff $n = 2^{p-1}(2^p - 1)$ and $2^p - 1$ is prime (Mersenne's). No odd perfect numbers are yet found.

## Carmichael numbers

A positive composite $n$ is a Carmichael number ($a^{n-1} \equiv 1 (mod\ n)$ for all $a : gcd(a, n) = 1$), iff $n$ is square-free, and for all prime divisors $p$ of $n$, $p - 1$ divides $n - 1$.

## Number/sum of divisors

```
//funcao sigma
//Soma dos divisores de n
int sigma(int n){
   int ans = 1;

   for (int i = 0; primes[i] * primes[i] <= n; i++){
      if (n % primes[i] == 0){
         int v = primes[i];

         while (n % primes[i] == 0){
            v *= primes[i];
            n /= primes[i];
         }

         ans *= (v - 1) / (primes[i] - 1);
      }
   }

   if (n > 1)
      ans *= n + 1;

   return ans;
}
```

$\tau(p_1^{a_1}...p_k^{a_k}) = \prod_{j=1}^{k}(a_j + 1)$.

$\sigma(p_1^{a_1}...p_k^{a_k}) = \prod_{j=1}^{k}\frac{p_j^{a_j+1}-1}{p_j-1}$.

## Euler's phi function

```
int phi(int n){
   int ans = n;

   for (int i = 0; primes[i] * primes[i] <= n; i++){
      if (n % primes[i] == 0){
         ans /= primes[i];
         ans *= primes[i] - 1;

         while(n % primes[i] == 0) n /= primes[i];
      }
   }

   if (n > 1){
      ans /= n;
      ans *= n - 1;
   }

   return ans;
}
```

$\phi(n) = |\{m \in \mathbb{N}, m \leq n, gcd(m,n) = 1\}|$.

$\phi(mn) = \frac{\phi(m)\phi(n)gcd(m,n)}{\phi(gcd(m,n))}$.

$\phi(p^a) = p^{a-1}(p-1)$.

$\sum_{d|n}\phi(d) = \sum_{d|n}\phi(\frac{n}{d}) = n$.

## Euler's Theorem

$a^{\phi(n)} \equiv 1(mod\ n)$, if $gcd(a,n) = 1$.

### Wilson's Theorem

$p$ is prime iff $(p-1)! \equiv -1 (mod\ p)$

### Mobius function

$\mu(1) = 1.$
$\mu(n) = 0$, if $n$ is not square free.
$\mu(n) = (-1)^s$, if $n$ is the product of s distinct primes.
Let $f$, $F$ be functions on positive integers. If for all $n \in \mathbb{N}$, $F(n) = \sum_{d|n} f(d)$, then $f(n) = \sum_{d|n} \mu(d) F(\frac{n}{d})$, and vice versa.
$\phi(n) = \sum_{d|n} \mu(d) \frac{n}{d}$.
$\sum_{d|n} \mu(d) = 1.$
If $f$ is multiplicative, then $\sum_{d|n} \mu(d) f(d) = \prod_{p|n}(1 - f(p))$, $\sum_{d|n} \mu(d)^2 f(d) = \prod_{p|n}(1 + f(p))$
f[i] = 1, f[p] = -1, f[j] *= (j%(i*i)==0) ? 0 : -1;

### Legendre symbol

If $p$ is an odd prime, $a \in \mathbb{Z}$, then $(\frac{a}{p})$ equals 0, if $p|a$; 1 if $a$ is a quadratic residue modulo $p$; and -1 otherwise. Euler's criterion: $(\frac{a}{p}) = a^{(\frac{p-1}{2})} (mod\ p)$.

### Jacobi symbol

If $n = p_1^{a_1} ... p_k^{a_k}$ is odd, then $(\frac{a}{n}) = \prod_{i=1}^{k} (\frac{a}{p_i})^{k_i}$.

### Primitive roots

If the order of $g$ modulo $m$ (min $n > 0 : g^n \equiv 1 (mod\ m)$) is $\phi(m)$, then $g$ is called a primitive root. If $\mathbb{Z}_m$ has a primitive root, then it has $\phi(\phi(m))$ distinct primitive roots. $\mathbb{Z}_m$ has a primitive root iff $m$ is one of $2, 4, p^k, 2p^k$, where $p$ is an odd prime. If $\mathbb{Z}_m$ has a primitive root $g$, then for all $a$ coprime to $m$, there exists unique integer $i = ind_g(a)$ modulo $\phi(m)$, such that $g^i \equiv a (mod\ m)$. $ind_g(a)$ has logarithm-like properties: $ind(1) = 0$, $ind(ab) = ind(a) + ind(b)$.
If $p$ is prime and $a$ is not divisible by $p$, then congruence $x^n \equiv a (mod\ p)$ has $gcd(n, p-1)$ solutions if $a^{(p-1)/gcd(n,p-1)} \equiv 1 (mod\ p)$, and no solutions otherwise.

### Discrete log

Find $x$ from $a^x \equiv b (mod\ m)$. Can be solved in $O(\sqrt{m})$ time and space with a meet-in-the-middle trick. Let $n = \lceil \sqrt{m} \rceil$, and $x = ny - z$. Equation becomes $a^{ny} \equiv ba^z (mod\ m)$. Precompute all values that the RHS can take for $z$ = 0,1,...,n-1, and brute force y on the LHS, each time checking whether there's a corresponding value for RHS.

### Pythagorean triples

Integer solutions of $x^2 + y^2 = z^2$. All relatively prime triples are given by: $x = 2mn, y = m^2 - n^2, z = m^2 + n^2$ where $m > n$, $gcd(m, n) = 1$ and $m \not\equiv n (mod\ 2)$. All other triples are multiples of these. Equation $x^2 + y^2 = 2z^2$ is equivalent to $(\frac{x+y}{2})^2 + (\frac{x-y}{2})^2 = z^2$.

### Postage stamps/McNuggets problem

Let $a, b$ be relatively-prime integers. There are exactly $\frac{1}{2}(a-1)(b-1)$ numbers *not* of form $ax+by (x, y \geq 0)$, and the largest is $(a-1)(b-1)-1 = ab-a-b$.

### Fermat's two-squares theorem

Odd prime $p$ can be represented as a sum of two squares iff $p \equiv 1 (mod\ 4)$. A product of two sums of two squares is a sum of two squares. Thus, $n$ is a sum of two squares iff every prime of form $p = 4k + 3$ occurs an even number of times in $n$'s factorization.

# String Algorithms

## String Hash

```
#define MAXN 10000
#define BASE 33ULL
#define VALUE(c) ((c)-'a')

typedef unsigned long long hash;

hash h[MAXN], pw[MAXN];

hash calc_hash(int beg, int end){
   return h[end] - h[beg]*pw[end-beg];
}

void init(){
   pw[0] = 1ULL;
   for (int i = 1; i < MAXN; ++i){
      pw[i] = pw[i-1]*BASE;
   }
   h[0] = 0ULL;
   for (int j = 0; s[j] != '\0'; ++j){
      h[j+1] = h[j]*BASE + VALUE(s[j]);
   }
}
```

## Prefix Function

```
void prefixfunction(string S){
   int N = SIZE(S);

   p[0] = p[1] = 0;
   FOR(i, 2, N){
      int j = p[i-1];
      while (S[i-1] != S[j]){
         if (j == 0){ j = -1; break; }
         j = p[j];
      }
      p[i] = ++j;
   }
}
```

## Z Function

```
void zfunction(string S){
   int N = SIZE(S), a = 0, b = 0;
   REP(i,N) z[i] = N;

   FOR(i, 1, N-1){
      int k = (i<b) ? min(b-i,z[i-a]) : 0;
      while (i+k < N && s[i+k]==s[k]) ++k;
      z[i] = k;
      if (i+k > b){ a = i; b = i+k; }
   }
}
```

## Manacher

```
void manacher(strind &ss){
   string s = "#";
   for (size_t i = 0, sz = ss.size(); i < sz; ++i){
      s += ss[i];
      s += "#";
   }

   int n = int(s.size());
   for (int i = 0; i < n; ++i) ans[i] = 0;

   int cur = 1;
   while (cur < n){

      while ((cur > ans[cur])
         && (cur+ans[cur]+1 < n)
         && (s[cur-ans[cur]-1] == s[cur+ans[cur]+1])) ans[cur]++;

      int j = 1;
      while ((cur+j < n) && (j < ans[cur]-ans[cur-j])){
         ans[cur+j] = ans[cur-j];
         j++;
      }

      if (cur+j < n)
         ans[cur+j] = ans[cur]-j;
      cur += j;
   }
}
```

## Aho-Corasick

```
struct No {
   int fail;
   vector< pair<int,int> > out; // num e tamanho do padrao
   //bool marc; // p/ decisao
   map<char, int> lista;
   int next; // aponta para o proximo sufixo que tenha out.size > 0
};
No arvore[1000003]; // quantida maxima de nos
//bool encontrado[1005]; // quantidade maxima de padroes, p/ decisao
int qtdNos, qtdPadroes;

// Funcao para inicializar
void inic() {
   arvore[0].fail = -1;
   arvore[0].lista.clear();
   arvore[0].out.clear();
   arvore[0].next = -1;
   qtdNos = 1;
   qtdPadroes = 0;
   //arvore[0].marc = false; // p/ decisao
   //memset(encontrado, false, sizeof(encontrado)); // p/ decisao
}

// Funcao para adicionar um padrao
void adicionar(char *padrao) {
   int no = 0, len = 0;
   for (int i = 0 ; padrao[i] ; i++, len++) {
      if (arvore[no].lista.find(padrao[i]) == arvore[no].lista.end()) {
         arvore[qtdNos].lista.clear(); arvore[qtdNos].out.clear();
         //arvore[qtdNos].marc = false; // p/ decisao
         arvore[no].lista[padrao[i]] = qtdNos;
         no = qtdNos++;
      } else no = arvore[no].lista[padrao[i]];
   }
   arvore[no].out.push_back(pair<int,int>(qtdPadroes++,len));
}

// Ativar Aho-corasick, ajustando funcoes de falha
void ativar() {
   int no,v,f,w;
   queue<int> fila;
   for (map<char,int>::iterator it = arvore[0].lista.begin();
       it != arvore[0].lista.end() ; it++) {
      arvore[no = it->second].fail = 0;
      arvore[no].next = arvore[0].out.size() ? 0 : -1;
      fila.push(no);
   }
   while (!fila.empty()) {
      no = fila.front(); fila.pop();
      for (map<char,int>::iterator it=arvore[no].lista.begin();
          it!=arvore[no].lista.end(); it++) {
         char c = it->first;
         v = it->second;
         fila.push(v);
         f = arvore[no].fail;
         while (arvore[f].lista.find(c) == arvore[f].lista.end()) {
            if (f == 0) { arvore[0].lista[c] = 0; break; }
            f = arvore[f].fail;
         }
         w = arvore[f].lista[c];
         arvore[v].fail = w;
         arvore[v].next = arvore[w].out.size() ? w : arvore[w].next;
      }
   }
}

// Buscar padroes no aho-corasik
void buscar(char *input) {
   int v, no = 0;
   for (int i = 0 ; input[i] ; i++) {
      while (arvore[no].lista.find(input[i]) == arvore[no].lista.end()) {
         if (no == 0) { arvore[0].lista[input[i]] = 0; break; }
         no = arvore[no].fail;
      }
      v = no = arvore[no].lista[input[i]];
      // marcar os encontrados
      while (v != -1 /* && !arvore[v].marc */ ) { // p/ decisao
         //arvore[v].marc = true; // p/ decisao: nao continua a lista
         for (int k = 0 ; k < arvore[v].out.size() ; k++) {
            //encontrado[arvore[v].out[k].first] = true; // p/ decisao
            printf("Padrao %d na posicao %d\n", arvore[v].out[k].first,
                  i-arvore[v].out[k].second+1);
         }
         v = arvore[v].next;
      }
   }
   // for (int i = 0 ; i < qtdPadroes ; i++)
   //printf("%s\n", encontrado[i]?"y":"n"); // p/ decisao
}
```

## Suffix Array O(n log n)

```
/* O( N log N ) SA build + O( N ) LCP build, #include <cstring> :P */
#define MAXN 100000
string S;
int N, SA[MAXN], LCP[MAXN], rank[MAXN], bucket[CHAR_MAX-CHAR_MIN+1];
char bh[MAXN+1];

void buildSA( bool needLCP = false ){
   int a, c, d, e, f, h, i, j, x;
   int *cnt = LCP;
   memset( bucket, -1, sizeof(bucket) );
   for ( i = 0; i < N; i++ ){
      j = S[i] - CHAR_MIN;
      rank[i] = bucket[j];
      bucket[j] = i;
   }
   for ( a = c = 0; a <= CHAR_MAX-CHAR_MIN; a++ ){
      for( i = bucket[a]; i != -1; i=j ){
         j = rank[i]; rank[i] = c;
         bh[c++] = (i == bucket[a]);
      }
   }
   bh[N] = 1;
   for ( i = 0; i < N; i++ )
      SA[ rank[i] ] = i;
   x = 0;
   for ( h = 1; h < N; h *= 2 ){
      for ( i = 0; i < N; i++ ){
         if ( bh[i] & 1 ){
            x = i;
            cnt[x] = 0;
         }
         rank[ SA[i] ] = x;
      }
      d = N-h; e = rank[d];
      rank[d] = e + cnt[e]++;
      bh[rank[d]] |= 2;
      i = 0;
      while ( i < N ){
         for ( j = i; (j == i || !(bh[j] & 1)) && j < N; j++ ){
            d = SA[j]-h;
            if ( d >= 0 ){
               e = rank[d]; rank[d] = e + cnt[e]++; bh[rank[d]] |= 2;
            }
         }
         for ( j = i; (j == i || !(bh[j] & 1)) && j < N; j++ ){
            d = SA[j]-h;
            if ( d >= 0 && (bh[rank[d]] & 2)){
               for ( e = rank[d]+1; bh[e] == 2; e++);
               for ( f = rank[d]+1; f < e; f++ ) bh[f] &= 1;
            }
         }
         i = j;
      }
      for ( i = 0; i < N; i++ ){
         SA[rank[i]] = i;
         if ( bh[i] == 2 ) bh[i] = 3;
      }
   }

   if ( needLCP ){
      LCP[0] = 0;
      for ( i = 0, h = 0; i < N; i++ ){
         e = rank[i];
         if ( e > 0 ){
            j = SA[e-1];
            while ( ((i+h) < N) && ((j+h) < N) && (S[i+h] == S[j+h]) ) h++;
            LCP[e] = h;
            if ( h > 0 ) h--;
         }
      }
   }
}
```

## Suffix Array O(n)

```
bool k_cmp(int a1, int b1, int a2, int b2, int a3 = 0, int b3 = 0){
   return a1 != b1 ? a1 < b1 : (a2 != b2 ? a2 < b2 : a3 < b3);
}

int bucket[MAXSZ+1], tmp[MAXSZ];
template<class T> void k_radix(T keys, int *in, int *out,
                        int off, int n, int k){
   memset(bucket, 0, sizeof(int) * (k+1));

   for (int j = 0; j < n; j++)
      bucket[keys[in[j]+off]]++;
   for (int j = 0, sum = 0; j <= k; j++)
      sum += bucket[j], bucket[j] = sum - bucket[j];
   for (int j = 0; j < n; j++)
      out[bucket[keys[in[j]+off]]++] = in[j];
}

int m0[MAXSZ/3+1];
vector<int> k_rec(const vector<int>& v, int k){
   int n = v.size()-3, sz = (n+2)/3, sz2 = sz + n/3;
   if (n < 2) return vector<int>(n);

   vector<int> sub(sz2+3);
   for (int i = 1, j = 0; j < sz2; i += i%3, j++)
      sub[j] = i;

   k_radix(v.begin(), &sub[0], tmp, 2, sz2, k);
   k_radix(v.begin(), tmp, &sub[0], 1, sz2, k);
   k_radix(v.begin(), &sub[0], tmp, 0, sz2, k);

   int last[3] = {-1, -1, -1}, unique = 0;
   for (int i = 0; i < sz2; i++){
      bool diff = false;
      for (int j = 0; j < 3; last[j] = v[tmp[i]+j], j++)
         diff |= last[j] != v[tmp[i]+j];
      unique += diff;

      if (tmp[i]%3 == 1) sub[tmp[i]/3] = unique;
      else sub[tmp[i]/3 + sz] = unique;
   }

   vector<int> rec;
   if (unique < sz2){
      rec = k_rec(sub, unique);
      rec.resize(sz2+sz);
      for (int i = 0; i < sz2; i++) sub[rec[i]] = i+1;
   }
   else{
      rec.resize(sz2+sz);
      for (int i = 0; i < sz2; i++) rec[sub[i]-1] = i;
   }

   for (int i = 0, j = 0; j < sz; i++)
      if (rec[i] < sz)
         tmp[j++] = 3*rec[i];
   k_radix(v.begin(), tmp, m0, 0, sz, k);
   for (int i = 0; i < sz2; i++)
      rec[i] = rec[i] < sz ? 3*rec[i] + 1 : 3*(rec[i] - sz) + 2;

   int prec = sz2-1, p0 = sz-1, pret = sz2+sz-1;
   while (prec >= 0 && p0 >= 0)
      if (rec[prec]%3 == 1 && k_cmp(v[m0[p0]], v[rec[prec]],
                           sub[m0[p0]/3], sub[rec[prec]/3+sz]) ||
        rec[prec]%3 == 2 && k_cmp(v[m0[p0]], v[rec[prec]],
                           v[m0[p0]+1], v[rec[prec]+1],
                           sub[m0[p0]/3+sz], sub[rec[prec]/3+1]))
         rec[pret--] = rec[prec--];
      else
         rec[pret--] = m0[p0--];
   if (p0 >= 0) memcpy(&rec[0], m0, sizeof(int) * (p0+1));

   if (n%3 == 1) rec.erase(rec.begin());
   return rec;
}

vector<int> karkkainen(const string& s){
   int n = s.size(), cnt = 1;
   vector<int> v(n + 3);

   for (int i = 0; i < n; i++) v[i] = i;
   k_radix(s.begin(), &v[0], tmp, 0, n, 256);
   for (int i = 0; i < n; cnt += (i+1 < n && s[tmp[i+1]] != s[tmp[i]]), i++)
      v[tmp[i]] = cnt;

   return k_rec(v, cnt);
}

vector<int> lcp(const string& s, const vector<int>& sa){
   int n = sa.size();
   vector<int> prm(n), ans(n-1);
   for (int i = 0; i < n; i++) prm[sa[i]] = i;

   for (int h = 0, i = 0; i < n; i++)
      if (prm[i]){
         int j = sa[prm[i]-1], ij = max(i, j);
         while (ij + h < n && s[i+h] == s[j+h]) h++;
         ans[prm[i]-1] = h;
         if (h) h--;
      }
   return ans;
}
```

## Suffix Automata

```
#define MAXN 250000

struct state_t{
   int len, link;
   map< char, int > next;

   bool clone;
   int first_pos;
   vector<int> inv_link;

   int cnt, nxt;
};

int sz, last;
state_t state[2*MAXN];

void automata_init(){
   sz = last = 0;
   state[0].len = 0;
   state[0].link = -1;
   ++sz;
}

void automata_extend(char c){
   int cur = sz++;
   state[cur].len = state[last].len+1;
   state[cur].first_pos = state[last].len;
   state[cur].cnt = 1;
   int p = last;

   for (; p != -1 && !state[p].next.count(c); p = state[p].link){
      state[p].next[c] = cur;
   }

   if (p == -1){
      state[cur].link = 0;
   }
   else{
      int q = state[p].next[c];
      if (state[p].len+1 == state[q].len){
         state[cur].link = q;
      }
      else{
         int clone = sz++;
         state[clone].len = state[p].len+1;
         state[clone].next = state[q].next;
         state[clone].link = state[q].link;
         state[clone].first_pos = state[q].first_pos;
         state[clone].clone = true;
         for (; p != -1 && state[p].next[c]==q; p=state[p].link){
            state[p].next[c] = clone;
         }
         state[q].link = state[cur].link = clone;
      }
   }
   last = cur;
}

for (int v = 1; v < sz; ++v)
   state[ state[v].link ].inv_link.push_back(v);

int first[n+1];
memset(first, -1, sizeof(first));
for (int v = 0; v < sz; ++v){
   state[v].nxt = first[state[v].len];
   first[state[v].len] = v;
}

for (int i = n; i >= 0; --i){
   for (int u = first[i]; u != -1; u = state[u].nxt){
      if (state[u].link != -1)
         state[ state[u].link ].cnt += state[u].cnt;
   }
}
```

- First occurrence of string P $= firstpos(v) - |P| + 1;$
- All occurrences: same as before, but must follow inverse suffix links and don't print clones.

### Burrows-Wheeler inverse transform

Let B[1..n] be the input (last column of sorted matrix of string's rotations). Get the first column, A[1...n], by sorting B. For each $k$-th occurrence of a character $c$ at index $i$ in A, let next[i] be the index of corresponding $k-th$ occurrence of $c$ in $B$. The $r$-th row of the matrix is A[r], A[next[r]], A[next[next[r]]], ...

### Huffman's algorithm

Start with a forest, consisting of isolated vertices. Repeatedly merge two trees with the lowest weights.

# Graphs

## Hopcroft-Karp

```
/* Maximum Bipartite Matching (Minimum Vertex Cover) on unweighted graph */
#define MAXN 111

int N, M; // N - # of vertexes on X, M - # of vertexes on Y
vector< int > gr[MAXN]; // gr[u] -- edges from u in X to v in Y
bool seen[MAXN];
int m[MAXN], m1[MAXN]; // with whom it's matched

int dfs(int u){
   if (u < 0) return 1;
   if (seen[u]) return 0;
   seen[u] = true;
   for (size_t i = 0, sz = gr[u].size(); i < sz; ++i) {
      if (dfs(m1[ gr[u][i] ])) {
         m[u] = gr[u][i];
         m1[ gr[u][i] ] = u;
         return 1;
      }
   }
   return 0;
}

int dfsExp(int u){
   for (int i = 0; i < N; ++i) seen[i] = false;
   return dfs(u);
}


int bipMatch(){
   for (int i = 0; i < N; ++i) m[i] = -1;
   for (int i = 0; i < M; ++i) m1[i] = -1;
   int aug, ans = 0;
   do{
      aug = 0;
      bool first = true;
      for (int i = 0; i < N; ++i) if (m[i] < 0){
         if (first) aug += dfsExp(i);
         else aug += dfs(i);
         first = false;
      }
      ans += aug;
   } while (aug);
   return ans;
}

/* needed for minium vertex cover.. */
int vx[MAXN], vy[MAXN];
void buildVC( int u ){
   seen[u] = true;
   vx[u] = 0;
   for (size_t w = 0, sz = gr[u].size(); w < sz; ++w)
      if (gr[u][w] != m[u] && vy[ gr[u][w] ] == 0){
         vy[ gr[u][w] ] = 1;
         if (!seen[ m1[ gr[u][w] ] ]) buildVC(m1[ gr[u][w] ]);
      }
}

// T ~ Unmatched L + reachable using alternating paths
// ANS .. (L \ T) U ( R intersect T )
for (int i = 0; i < N; ++i) {
   seen[i] = false;
   if (m[i] == -1) vx[i] = 0; // T -- unmatched L
   else vx[i] = 1; // L \ T -- for now...
}

for (int i = 0; i < M; ++i) vy[i] = 0; // R .. ~T -- for now..
for (int i = 0; i < N; ++i) if (vx[i] == 0 && !seen[i]) buildVC(i);
```

For any graph, if it has an even cycle, this graph is bipartite

### Euler's theorem

For any planar graph, $V - E + F = 1 + C$, where $V$ is the number of graph's vertices, $E$ is the number of edges, $F$ is the number of faces in graph's planar drawing, and $C$ is the number of connected components. Corollary: $V - E + F = 2$ for a 3D polyhedron.

### Vertex covers and independent sets

Let $M$, $C$, $I$ be a max matching, a min vertex cover, and a max independent set. Then $|M| \leq |C| = N - |I|$, with equality for bipartite graphs. Complement of an MVC is always a MIS, and vice versa. Given a bipartite graph with partitions $(A, B)$, build a network: connect source to $A$, and $B$ to sink with edges of capacities, equal to the corresponding nodes' weights, or 1 in the unweighted case. Set capacities of the original graph's edges to the infinity. Let $(S, T)$ be a minimum $s - t$ cut. Then a maximum(-weighted) independent set is $I = (A \bigcap S) \bigcup (B \bigcap T)$, and a minimum(-weighted) vertex cover is $C = (A \bigcap T) \bigcup (B \bigcap S)$.

### Matrix-tree theorem

Let matrix $T = [t_{ij}]$, where $t_{ij}$ is the number of multiedges between $i$ and $j$, for $i \neq j$, and $t_{ii} = -deg_i$. Number of spanning trees of a graph is equal to the determinant of a matrix obtained by deleting any $k$-th row and $k$-th column from $T$.

### Prufer code of a tree

Label vertices with integers 1 to $n$. Repeatedly remove the leaf with the smallest label, and output its only neighbor's label, until only one edge remains. The sequence has length $n-2$. Two isomorphic trees have the same sequence, and every sequence of integers from 1 and $n$ corresponds to a tree. Corollary: the number of labelled trees with $n$ vertices is $n^{n-2}$.

### Euler tours

Euler tour in an undirected graph exists iff the graph is connected and each vertex has an even degree. Euler tour in a directed graph exists iff in-degree of each vertex equals its out-degree, and underlying undirected graph is connected. Construction:

```
doit(u):
  for each edge e = (u, v) in E, do: erase e, doit(v)
  prepend u to the list of vertices in the tour
```

### Stable marriage problem

While there is a free man $m$: let $w$ be the most-preferred woman to whom he has not yet proposed, and propose $m$ to $w$. If $w$ is free, or is engaged to someone whom she prefers less than $m$, match $m$ with $w$, else deny proposal.

```
int prefList[430][430];
int status[830]; /* status[i] contains husband/wife of i, initially -1 */
map<int, string> rev_bib;

void stableMarriage(int n){
   FOR(i, 2*n) status[i] = -1; /* 0...n mens, n...2*n women */
   queue<int> singleM;
   FOR(i, n) singleM.push(i); /* Push all single men */

   /* While there is a single men */
   while (!singleM.empty()){
      int i = singleM.front();
      singleM.pop();
      FOR(j, n){
         /* if girl is single marry her to this man */
         int singleW = prefList[i][j];
         if (status[singleW] == -1){
            status[i] = singleW; /* set this girl as wife of i */
            status[singleW] = i; /*make i as husband of this girl*/
            break;
         }
         else{
            int rank1, rank2; /* for holding priority of current */
            FOR(k, n){ /* husband and most preferable husband */
               if (prefList[singleW][k] == status[singleW]) rank1 = k;
               if (prefList[singleW][k] == i) rank2 = k;
            }
            /* if this girl j prefers current man i more than her present husband */
            if (rank2 < rank1){
               status[i] = singleW; /* her wife of i */
               int old = status[singleW];
               status[old] = -1; /* divorce current husband */
               singleM.push(old); /* add him to list of singles */
               status[singleW] = i; /* set new husband for this girl */
               break;
            }
         }
      }
   }

   FOR(i, n){
      /* print each matching */
      cout << rev_bib[i] << " " << rev_bib[status[i]] << endl;
   }
}
```

## Stoer-Wagner

Start from a set $A$ containing an arbitrary vertex. While $A \neq V$, add to $A$ the most tightly connected vertex ($z \notin A$ such that $\sum_{x \in A} w(x, z)$ is maximized.) Store cut-of-the-phase (the cut between the last added vertex and rest of the graph), and merge the two vertices added last. Repeat until the graph is contracted to a single vertex. Minimum cut is one of the cuts-of-the-phase.

```
/* Stoer-Wagner Min Cut on undirected graph */
#define MAXV 101

int grafo[MAXV][MAXV];

// v[i] representa o vertice original do grafo correspondente
// ao i-esimo vertice do grafo da fase atual do minCut e w[i]
// tem o peso do vertice v[i]..
int v[MAXV], w[MAXV];
int A[MAXV];

int minCut(int n){
   if (n == 1) return 0;

   int i, u, x, s, t;
   int minimo;

   for (u = 1; u <= n; ++u){ v[u] = u; }

   w[0] = -1;
   minimo = INF;

   while (n > 1){

      for (u = 1; u <= n; ++u){
         A[v[u]] = 0;
         w[u] = grafo[v[1]][v[u]];
      }

      A[v[1]] = 1;
      s = v[1];
      for (u = 2; u <= n; ++u){

         // Encontra o mais fortemente conetado a A
         t = 0;
         for (x = 2; x <= n; ++x)
            if (!A[v[x]] && (w[x] > w[t]))
               t = x;

         // adiciona ele a A
         A[v[t]] = 1;

         if (u == n){
            if (w[t] < minimo)
               minimo = w[t];

            // Une s e t
            for (x = 1; x <= n; ++x){
               grafo[s][v[x]] += grafo[v[t]][v[x]];
               grafo[v[x]][s] = grafo[s][v[x]];
            }

            v[t] = v[n--];
            break;
         }
         s = v[t];

         // Atualiza os pesos
         for (x = 1; x <= n; ++x)
            w[x] += grafo[v[t]][v[x]];
      }
   }
   return minimo;
}
```

### Tarjan's offline LCA algorithm

```
DFS(x):
  ancestor[Find(x)] = x
  for all children y of x:
     DFS(y); Union(x, y); ancestor[Find(x)] = x
  seen[x] = true
  for all queries {x, y}:
    if seen[y] then output "LCA(x, y) is ancestor[Find(y)]"
```

### Erdos-Gallai theorem

A sequence of integers $\{d_1, d_2, ..., d_n\}$, with $n-1 \geq d_1 \geq d_2 \geq ... \geq d_n \geq 0$ is a degree sequence of some undirected simple graph iff $\sum d_i$ is even and $d_1 + ... + dk \leq k(k-1) + \sum_{i=k+1}^{n} min(k, d_i)$ for all $k = 1, 2, ..., n-1$.

### Dilworth's theorem

In any finite partially ordered set, the maximum number of elements in any antichain equals the minimum number of chains in any partition of the set into chains.

Dilworth's theorem characterizes the width of any finite partially ordered set in terms of a partition of the order into a minimum number of chains. An antichain in a partially ordered set is a set of elements no two of which are comparable to each other, and a chain is a set of elements every two of which are comparable. Dilworth's theorem states that there exists an antichain $A$, and a partition of the order into a family $P$ of chains, such that the number of chains in the partition equals the cardinality of $A$. When this occurs, $A$ must be the largest antichain in the order, for any antichain can have at most one element from each member of $P$. Similarly, $P$ must be the smallest family of chains into which the order can be partitioned, for any partition into chains must have at least one chain per element of $A$. The width of the partial order is defined as the common size of $A$ and $P$.

## Tarjan algorithm for articulation points

```
#define MAXN 111

int N;

vector< int > gr[ MAXN ];
int low[MAXN], lbl[MAXN], parent[MAXN];
int dfsnum;
int rchild; // child count of the root
int root; // root of the tree
int arts; // # of critical vertexes
bool art[MAXN];

void dfs( int u ){
   lbl[u] = low[u] = dfsnum++;
   for ( size_t i = 0, sz = gr[u].size(); i < sz; i++ ){
      int v = gr[u][i];
      if ( lbl[v] == -1 ){
         parent[v] = u;
         if ( u == root ) rchild++;
         dfs( v );
         if ( u != root && low[v] >= lbl[u] && !art[u] ) art[u] = true, arts++;
         low[u] = min( low[u], low[v] );
      }
      else if( v != parent[u] ) low[u] = min( low[u], lbl[v] );
   }
   if ( u == root && rchild > 1 ) art[u] = true, arts++;
}
```

## Tarjan algorithm for brigdes

```
#define MAXN 111

int N;
vector< int > gr[MAXN];
int low[MAXN], lbl[MAXN], parent[MAXN];
int dfsnum;
vector< pair<int,int> > brid; // the bridges themselves

void dfs( int u ){
   lbl[u] = low[u] = dfsnum++;
   for ( size_t i = 0, sz = gr[u].size(); i < sz; i++ ){
      int v = gr[u][i];
      if ( lbl[v] == -1 ){
         parent[v] = u;
         dfs( v );
         if ( low[v] > lbl[u] )
            brid.push_back( make_pair(u, v) );
         low[u] = min( low[u], low[v] );

         /*
         if( low[u] > low[v] ) low[u] = low[v];
         if( low[v] == lbl[v] )
            brid.push_back( make_pair(u, v) );
         */
      }
      else if( v != parent[u] ) low[u] = min( low[u], lbl[v] );
   }
}
```

## Tarjan algorithm for strongly connected components

```
#define MAXN 111

int N;

int low[MAXN], lbl[MAXN], dfsnum;
vector<int> gr[MAXN];
bool stkd[MAXN];
stack< int > scc;

void dfs( int u ){
   lbl[u] = low[u] = dfsnum++;
   scc.push( u );
   stkd[u] = true;
   int v;
   for ( size_t i = 0, sz = gr[u].size(); i < sz; i++ ){
      v = gr[u][i];
      if( lbl[v] == -1 ) dfs( v );
      if( stkd[v] ) low[u] = min( low[u], low[v] );
   }
   if ( low[u] == lbl[u] ){ // new component found...
      while( !scc.empty() && scc.top() != u ){
         // ...with these guys
         stkd[ scc.top() ] = false;
         scc.pop();
      }
      scc.pop(); stkd[u] = false;
   }
}
```

## Floyd-Warshall reconstructing path

```
/*
   init: p[i][j] = i;
   if (i,k)+(k,j) < (i,j)
    p[i][j] = p[k][j]
*/

void show( int from, int to ){
   if( from != to ){
      show( from, p[from][to] );
      cout << " ";
   }
   cout << to;
}
```

## Busacker Gowen

```
int dist[MAXV], last_edge[MAXV], d_visited[MAXV], bg_prev[MAXV], pot[MAXV],
   capres[MAXV];
int prev_edge[MAXE], adj[MAXE], cap[MAXE], cost[MAXE], flow[MAXE];

int nedges;
priority_queue<pair<int, int> > d_q;

void bg_edge(int v, int w, int capacity, int cst, bool r = false){
   prev_edge[nedges] = last_edge[v];
   adj[nedges] = w;
   cap[nedges] = capacity;
   flow[nedges] = 0;
   cost[nedges] = cst;
   last_edge[v] = nedges++;

   if (!r) bg_edge(w, v, 0, -cst, true);
}

int rev(int i){ return i ^ 1; }
int from(int i){ return adj[rev(i)]; }

void bg_init(){
   nedges = 0;
   memset(last_edge, -1, sizeof last_edge);
   memset(pot, 0, sizeof pot);
}

void bg_dijkstra(int s, int num_nodes = MAXV){
   memset(dist, 0x3f, sizeof dist);
   memset(d_visited, 0, sizeof d_visited);
   d_q.push(make_pair(dist[s] = 0, s));
   capres[s] = 0x3f3f3f3f;

   while (!d_q.empty()){
      int v = d_q.top().second; d_q.pop();
      if (d_visited[v]) continue; d_visited[v] = true;

      for (int i = last_edge[v]; i != -1; i = prev_edge[i]){
         if (cap[i] - flow[i] == 0) continue;
         int w = adj[i], new_dist = dist[v] + cost[i] + pot[v] - pot[w];

         if (new_dist < dist[w]){
            d_q.push(make_pair(-(dist[w] = new_dist), w));
            bg_prev[w] = rev(i);
            capres[w] = min(capres[v], cap[i] - flow[i]);
         }
      }
   }
}

pair<int, int> busacker_gowen(int src, int sink, int num_nodes = MAXV){
   int ret_flow = 0, ret_cost = 0;

   bg_dijkstra(src, num_nodes);
   while (dist[sink] < 0x3f3f3f3f){
      int cur = sink;
      while (cur != src){
         flow[bg_prev[cur]] -= capres[sink];
         flow[rev(bg_prev[cur])] += capres[sink];
         ret_cost += cost[rev(bg_prev[cur])] * capres[sink];
         cur = adj[bg_prev[cur]];
      }
      ret_flow += capres[sink];

      for (int i = 0; i < MAXV; ++i)
         pot[i] = min(pot[i] + dist[i], 0x3f3f3f3f);

      bg_dijkstra(src, num_nodes);
   }
   return make_pair(ret_flow, ret_cost);
}
```

## Dinic

```
int last_edge[MAXV], cur_edge[MAXV], dist[MAXV];
int prev_edge[MAXE], cap[MAXE], flow[MAXE], adj[MAXE];
int nedges;

void d_init(){
   nedges = 0;
   memset(last_edge, -1, sizeof last_edge);
}

void d_edge(int v, int w, int capacity, bool r = false){
   prev_edge[nedges] = last_edge[v];
   cap[nedges] = capacity;
   adj[nedges] = w;
   flow[nedges] = 0;
   last_edge[v] = nedges++;

   if (!r) d_edge(w, v, 0, true);
}

bool d_auxflow(int source, int sink){
   queue<int> q;
   q.push(source);

   memset(dist, -1, sizeof dist);
   dist[source] = 0;
   memcpy(cur_edge, last_edge, sizeof last_edge);

   while (!q.empty()){
      int v = q.front(); q.pop();
      for (int i = last_edge[v]; i != -1; i = prev_edge[i]){
         if (cap[i] - flow[i] == 0) continue;

         if (dist[adj[i]] == -1){
            dist[adj[i]] = dist[v] + 1;
            q.push(adj[i]);
            if (adj[i] == sink) return true;
         }
      }
   }

   return false;
}

int d_augmenting(int v, int sink, int c){
   if (v == sink) return c;

   for (int& i = cur_edge[v]; i != -1; i = prev_edge[i]){
      if (cap[i] - flow[i] == 0 || dist[adj[i]] != dist[v] + 1)
         continue;

      int val;
      if (val = d_augmenting(adj[i], sink, min(c, cap[i] - flow[i]))){
         flow[i] += val;
         flow[i^1] -= val;
         return val;
      }
   }

   return 0;
}

int dinic(int source, int sink){
   int ret = 0;
   while (d_auxflow(source, sink)){
      int flow;
      while (flow = d_augmenting(source, sink, 0x3f3f3f3f))
         ret += flow;
   }

   return ret;
}
```

## Edmonds Karp

```
int last_edge[MAXV], ek_visited[MAXV], ek_prev[MAXV], ek_capres[MAXV];
int prev_edge[MAXE], cap[MAXE], flow[MAXE], adj[MAXE], nedges;

void ek_init(){
   nedges = 0;
   memset(last_edge, -1, sizeof last_edge);
}

void ek_edge(int v, int w, int capacity, bool r = false){
   prev_edge[nedges] = last_edge[v];
   cap[nedges] = capacity;
   adj[nedges] = w;
   flow[nedges] = 0;
   last_edge[v] = nedges++;
   if(!r) ek_edge(w, v, 0, true);
}

queue<int> ek_q;

inline int rev(int i){ return i ^ 1; }

int ek_bfs(int src, int sink, int num_nodes){
   memset(ek_visited, 0, sizeof(int) * num_nodes);

   ek_q = queue<int>();
   ek_q.push(src);
   ek_capres[src] = 0x3f3f3f3f;

   while (!ek_q.empty()){
      int v = ek_q.front(); ek_q.pop();
      if (v == sink) return ek_capres[sink];
      ek_visited[v] = 2;

      for (int i = last_edge[v]; i != -1; i = prev_edge[i]){
         int w = adj[i], new_capres = min(cap[i] - flow[i], ek_capres[v]);
         if (new_capres <= 0) continue;

         if (!ek_visited[w]){
            ek_prev[w] = rev(i);
            ek_capres[w] = new_capres;

            ek_visited[w] = 1;
            ek_q.push(w);
         }
      }
   }
   return 0;
}

int edmonds_karp(int src, int sink, int num_nodes = MAXV){
   int ret = 0, new_flow;

   while ((new_flow = ek_bfs(src, sink, num_nodes)) > 0){
      int cur = sink;
      while (cur != src) {
         flow[ek_prev[cur]] -= new_flow;
         flow[rev(ek_prev[cur])] += new_flow;
         cur = adj[ek_prev[cur]];
      }
      ret += new_flow;
   }
   return ret;
}
```

## Gabow (Maximum weighted matching in general graghs)

```
int prev_edge[MAXE], v[MAXE], w[MAXE], last_edge[MAXV];
int type[MAXV], label[MAXV], first[MAXV], mate[MAXV], nedges;
bool g_flag[MAXV], g_souter[MAXV];

void g_init(){
   nedges = 0;
   memset(last_edge, -1, sizeof last_edge);
}

void g_edge(int a, int b, bool rev = false){
   prev_edge[nedges] = last_edge[a];
   v[nedges] = a;
   w[nedges] = b;
   last_edge[a] = nedges++;

   if (!rev) return g_edge(b, a, true);
}

void g_label(int v, int join, int edge, queue<int>& outer){
   if (v == join) return;
   if (label[v] == -1) outer.push(v);

   label[v] = edge;
   type[v] = 1;
   first[v] = join;

   g_label(first[label[mate[v]]], join, edge, outer);
}

void g_augment(int _v, int _w){
   int t = mate[_v];
   mate[_v] = _w;

   if (mate[t] != _v) return;
   if (label[_v] == -1) return;

   if (type[_v] == 0){
      mate[t] = label[_v];
      g_augment(label[_v], t);
   }
   else if (type[_v] == 1){
      g_augment(v[label[_v]], w[label[_v]]);
      g_augment(w[label[_v]], v[label[_v]]);
   }
}

int gabow(int n){
   memset(mate, -1, sizeof mate);
   memset(first, -1, sizeof first);

   int ret = 0;
   for (int z = 0; z < n; ++z){
      if (mate[z] != -1) continue;

      memset(label, -1, sizeof label);
      memset(type, -1, sizeof type);
      memset(g_souter, 0, sizeof g_souter);

      label[z] = -1; type[z] = 0;

      queue<int> outer;
      outer.push(z);

      bool done = false;
      while (!outer.empty()){
         int x = outer.front(); outer.pop();

         if (g_souter[x]) continue;
         g_souter[x] = true;

         for (int i = last_edge[x]; i != -1; i = prev_edge[i]){
            if (mate[w[i]] == -1 && w[i] != z){
               mate[w[i]] = x;
               g_augment(x, w[i]);
               ++ret;

               done = true;
               break;
            }

            if (type[w[i]] == -1){
               int v = mate[w[i]];
               if (type[v] == -1){
                  type[v] = 0;
                  label[v] = x;
                  outer.push(v);

                  first[v] = w[i];
               }
               continue;
            }

            int r = first[x], s = first[w[i]];
            if (r == s) continue;

            memset(g_flag, 0, sizeof g_flag);
            g_flag[r] = g_flag[s] = true;
```

```
      while (r != -1 || s != -1) {
         if (s != -1) swap(r, s);
         r = first[label[mate[r]]];
         if (r == -1) continue;
         if (g_flag[r]) break; g_flag[r] = true;
      }

      g_label(first[x], r, i, outer);
      g_label(first[w[i]], r, i, outer);
```

```
               for (int c = 0; c < n; ++c)
                  if (type[c] != -1 && first[c] != -1 && type[first[c]] != -1)
                     first[c] = r;
            }
            if (done) break;
         }
      }
   return ret;
}
```

## Union Find

```
struct no{
   int pai, rank;
};

typedef struct no UJoin;

UJoin pset[MAXV];

void initialize(){
   for ( int i = 0; i < V; ++i ){
      pset[i].pai = i;
      pset[i].rank = visited[i] = 0;
      dfs_parent[i] = dfs_low[i] = dfs_num[i] = 0;
      directed[i].clear(); undirected[i].clear();
   }
}

void link (int x, int y){
   if ( pset[x].rank > pset[y].rank ) pset[y].pai = x;
   else{
      pset[x].pai = y;
      if ( pset[x].rank == pset[y].rank )
         pset[y].rank = pset[y].rank + 1;
   }
}

int findSet ( int x ){
   while ( pset[x].pai != x )
      x = pset[x].pai;
   return x;
}

void unionSet ( int x, int y ){
   link ( findSet(x), findSet(y) );
}

bool isSameSet ( int x, int y ){
   return findSet(x) == findSet(y);
}
```

## Gomory Hu Tree

```
bool cut[MAXV];
int mincut(int s, int t){
   memset(flow, 0, sizeof flow);
   memset(cut, 0, sizeof cut);
   int ret = dinic(s, t);

   queue<int> q;
   q.push(s); cut[s] = true;
   while (!q.empty()){
      int v = q.front(); q.pop();
      for (int i = last_edge[v]; i != -1; i = prev_edge[i]){
         int w = adj[i];
         if (cap[i] - flow[i] && !cut[w])
            cut[w] = true, q.push(w);
      }
   }

   return ret;
}

int up[MAXV], val[MAXV];
void gomory_hu(int n) {
   memset(up, 0, sizeof up);
   for (int i = 1; i < n; i++){
      val[i] = mincut(i, up[i]);
      for (int j = i+1; j < n; j++)
         if (cut[j] && up[j] == up[i])
            up[j] = i;
   }
}
```

## Kuhn Munkres (Maximum weighted bipartite matching)

```
int w[MAXV][MAXV], s[MAXV], rem[MAXV], remx[MAXV];
int mx[MAXV], my[MAXV], lx[MAXV], ly[MAXV];

void add(int x, int n){
   s[x] = true;
   for (int y = 0; y < n; y++)
      if (rem[y] != -INF && rem[y] > lx[x] + ly[y] - w[x][y])
         rem[y] = lx[x] + ly[y] - w[x][y], remx[y] = x;
}

int kuhn_munkres(int n){
   for (int i = 0; i < n; i++) mx[i] = my[i] = -1, lx[i] = ly[i] = 0;
   for (int i = 0; i < n; i++)
      for (int j = 0; j < n; j++)
         ly[j] = max(ly[j], w[i][j]);

   for (int i = 0; i < n; i++){
      memset(s, 0, sizeof s); memset(rem, 0x3f, sizeof rem);

      int st;
      for (st = 0; st < n; st++) if (mx[st] == -1){ add(st, n); break; }
      while (mx[st] == -1){
         int miny = -1;
         for (int y = 0; y < n; y++)
            if (rem[y] != -INF && (miny == -1 || rem[miny] >= rem[y]))
               miny = y;

         if (rem[miny]){
            for (int x = 0; x < n; x++) if (s[x]) lx[x] -= rem[miny];
            for (int y = 0, d = rem[miny]; y < n; y++)
               if (rem[y] == -INF) ly[y] += d; else rem[y] -= d;
         }

         if (my[miny] == -1){
            int cur = miny;
            while (remx[cur] != st){
               int pmate = mx[remx[cur]];
               my[cur] = remx[cur], mx[my[cur]] = cur;
               my[pmate] = -1; cur = pmate;
            }
            my[cur] = remx[cur], mx[my[cur]] = cur;
         }
         else add(my[miny], n), rem[miny] = -INF;
      }
   }

   int ret = 0;
   for (int i = 0; i < n; i++)
      ret += w[i][mx[i]];
   return ret;
}
```

## Link Cut Tree

```
class splay{
   public:
      splay *sons[2], *up, *path_up;
      splay() : up(NULL), path_up(NULL){
         sons[0] = sons[1] = NULL;
      }

      bool is_r(splay* n) {
         return n == sons[1];
      }
};

void rotate(splay* t, bool to_l){
   splay* n = t->sons[to_l]; swap(t->path_up, n->path_up);
   t->sons[to_l] = n->sons[!to_l]; if (t->sons[to_l]) t->sons[to_l]->up = t;
   n->up = t->up; if (n->up) n->up->sons[n->up->is_r(t)] = n;
   n->sons[!to_l] = t; t->up = n;
}

void do_splay(splay* n){
   for (splay* p; (p = n->up) != NULL; )
      if (p->up == NULL)
         rotate(p, p->is_r(n));
      else{
         bool dirp = p->is_r(n), dirg = p->up->is_r(p);
         if (dirp == dirg)
            rotate(p->up, dirg), rotate(p, dirp);
         else
            rotate(p, dirp), rotate(n->up, dirg);
      }
}

struct link_cut{
   splay* vtxs;
   link_cut(int numv){ vtxs = new splay[numv]; }
   ~link_cut(){ delete[] vtxs; }

   void access(splay* ov){
      for (splay *w = ov, *v = ov; w != NULL; v = w, w = w->path_up){
         do_splay(w);
         if (w->sons[1]) w->sons[1]->path_up = w, w->sons[1]->up = NULL;
         if (w != v) w->sons[1] = v, v->up = w, v->path_up = NULL;
         else w->sons[1] = NULL;
      }
      do_splay(ov);
   }

   splay* find(int v){
      splay* s = &vtxs[v];
      access(s); while (s->sons[0]) s = s->sons[0]; do_splay(s);
      return s;
   }

   void link(int parent, int son){
      access(&vtxs[son]); access(&vtxs[parent]);
      assert(vtxs[son].sons[0] == NULL);
      vtxs[son].sons[0] = &vtxs[parent];
      vtxs[parent].up = &vtxs[son];
   }

   void cut(int v){
      access(&vtxs[v]);
      if (vtxs[v].sons[0]) vtxs[v].sons[0]->up = NULL;
      vtxs[v].sons[0] = NULL;
   }

   int lca(int v, int w){
      access(&vtxs[v]); access(&vtxs[w]); do_splay(&vtxs[v]);
      if (vtxs[v].path_up == NULL) return v;
      return vtxs[v].path_up - vtxs;
   }
};
```

## Heavy Light Decomposition

```
vector<int> gr[MAXN];

int depth[MAXN], parent[MAXN], treesz[MAXN];
int chain[MAXN], chainpos[MAXN], chainleader[MAXN];

int N, cur_chain, pos;

void explore(int u){
   int v;
   treesz[u] = 1;
   for (size_t i = 0, sz = gr[u].size(); i < sz; ++i){
      v = gr[u][i];
      if (parent[ v ] == -1){
         parent[ v ] = u;
```

```
            depth[ v ] = depth[ u ]+1;
            explore(v);
            treesz[ u ] += treesz[ v ];
         }
      }
}

void decompose(int u, bool light = true){
   if (light) {
      ++cur_chain;
      chainleader[ cur_chain ] = u;
   }

   chain[ u ] = cur_chain;
   chainpos[ u ] = pos++;

   int v, ind = -1, mx = -1;
   for ( size_t i = 0, sz = gr[u].size(); i < sz; i++ ){
      v = gr[u][i];
      if (parent[ v ] == u && (mx == -1 || treesz[mx] < treesz[v]))
         mx = v, ind = i;
   }
   if (mx != -1){
      decompose(mx, false);
   }

   for (size_t i = 0, sz = gr[u].size(); i < sz; ++i){
      v = gr[u][i];
      if (parent[ v ] == u && v != mx){
         decompose( v );
      }
   }
}

int lca(int u, int v){
   while (chain[u] != chain[v]){
      if (depth[ chainleader[chain[u]] ] < depth[ chainleader[chain[v]] ])
         v = parent[ chainleader[ chain[v] ] ];
      else
         u = parent[ chainleader[ chain[u] ] ];
   }
   if (depth[u] < depth[v]) return u;
   return v;
}
```

## LCA / PD

```
void process(){
   int i, j;

   for (i = 0; i < N; i++)
      for (j = 0; 1 << j < N; j++)
         P[i][j] = -1;

   for (i = 0; i < N; i++)
      P[i][0] = T[i];

   for (j = 1; 1 << j < N; j++)
      for (i = 0; i < N; i++)
         if (P[i][j - 1] != -1)
            P[i][j] = P[P[i][j - 1]][j - 1];
}

int query(int p, int q){
   int tmp, log, i;

   //if p is situated on a higher level than q then we swap them
   if (L[p] < L[q])
      tmp = p, p = q, q = tmp;

   //we compute the value of [log(L[p)]
   for (log = 1; 1 << log <= L[p]; log++);
   log--;

   //we find the ancestor of node p situated on the same level
   //with q using the values in P
   for (i = log; i >= 0; i--)
      if (L[p] - (1 << i) >= L[q])
         p = P[p][i];

   if (p == q)
      return p;

   //we compute LCA(p, q) using the values in P
   for (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 T[p];
}
```

## Fast Dijkstra

```
struct el { int n, d; } heap[MAXV]; int where[MAXV];
int prev_edge[MAXE], weight[MAXE], adj[MAXE], dist[MAXV], last_edge[MAXV];
int nedges;

void d_init() {
   nedges = 0;
   memset(last_edge, -1, sizeof last_edge);
}

void d_edge(int v, int w, int eweight) {
   prev_edge[nedges] = last_edge[v];
   weight[nedges] = eweight;
   adj[nedges] = w;
   last_edge[v] = nedges++;
}

void d_swap(int a, int b) {
   swap(heap[a], heap[b]);
   swap(where[heap[a].n], where[heap[b].n]);
}

void decrease_key(int n, int d) {
   heap[where[n]].d = d;
   for(int cur = where[n], up; cur && d < heap[up = (cur-1)/2].d; cur = up)
      d_swap(cur, up);
}

int pop_heap(int& sz) {
   int cur = 0;
   for(int next = 2; next < sz; cur = next, next = 2*next+2) {
      if(heap[next].d > heap[next-1].d) next--;
      d_swap(next, cur);
   }
   d_swap(cur, sz-1);
   return heap[--sz].n;
}

void dijkstra(int s, int sz = MAXV) {
   for(int i = 0; i < sz; ++i)
      heap[i].n = where[i] = i, heap[i].d = dist[i] = 0x3f3f3f3f;
   decrease_key(s, dist[s] = 0);

   while(sz)
      for(int v = pop_heap(sz), i = last_edge[v]; i != -1; i = prev_edge[i]) {
         int w = adj[i];
         if(dist[v] + weight[i] < dist[w])
            decrease_key(w, dist[w] = dist[v] + weight[i]);
      }
}
```

# Data structures

## Treap

```
struct seg_info{
   // ...
   seg_info() {}
   void merge (seg_info* left, int key, seg_info* right){
      // ...
   }
};

struct node_t{
   int key, pr, sz;
   seg_info seg;

   node_t *l, *r;
   node_t(int k) : key(k), pr(rand()), sz(0), l(NULL), r(NULL) {}
   /*~node_t(){
      if(l) delete l;
      if(r) delete r;
   }*/
};

void rotate_right(node_t* &t){
   node_t *n = t->l;
   t->l = n->r;
   n->r = t;
   t = n;
}

void rotate_left(node_t* &t){
   node_t *n = t->r;
   t->r = n->l;
   n->l = t;
   t = n;
}

void fix(node_t* t){
   if (!t) return;
   t->sz = 1 + ((t->l)?(t->l->sz):(0)) + ((t->r)?(t->r->sz):(0));
   seg_info *lseg, *rseg;
   lseg = (t->l)?(&(t->l->seg)):(NULL);
   rseg = (t->r)?(&(t->r->seg)):(NULL);
   t->seg.merge(lseg, t->key, rseg);
}

void insert(node_t* &t, int val, int pos){
   if (!t) t = new node_t(val);
   else{
      int lsz = ((t->l)?(t->l->sz):(0));
      if (lsz >= pos) insert(t->l, val, pos);
      else insert(t->r, val, pos-lsz-1);
   }

   if (t->l && ((t->l->pr) > (t->pr))) rotate_right(t);
   else if (t->r && ((t->r->pr) > (t->pr))) rotate_left(t);

   fix(t->l); fix(t->r); fix(t);
}

inline int p(node_t* t){ return (t) ? (t->pr) : (-1); }

void erase(node_t* &t, int pos){
   if (!t) return;

   int lsz = ((t->l)?(t->l->sz):(0));
   if (lsz+1 != pos){
      if (lsz >= pos) erase(t->l, pos);
      else erase(t->r, pos-lsz-1);
   }
   else{
      if (!t->l && !t->r){
         //delete t;
         t = NULL;
      }
      else{
         if (p(t->l) < p(t->r)) rotate_left(t);
         else rotate_right(t);
         erase( t, pos );
      }
   }

   if (t){ fix(t->l); fix(t->r); } fix(t);
}

void replace(node_t* t, int pos, int val){
   if (!t) return;

   int lsz = ((t->l) ? (t->l->sz) : (0));
   if (lsz+1 != pos){
      if (lsz >= pos) replace(t->l, pos, val);
      else replace(t->r, pos-lsz-1, val);
   }
   else t->key = val;

   fix(t);
}
```

```
seg_info query(node_t* t, int lo, int hi){
   if (x <= lo && hi <= y){
      return t->seg;
   }
   int mid = lo + ((t->l) ? (t->l->sz) : (0)) - 1;

   seg_info q1, q2, ans;
   bool f1, f2; f1 = f2 = false;
   if (mid >= lo && mid >= x){ f1 = true; q1 = query(t->l, lo, mid); }
   if (mid+2 <= hi && mid+2 <= y){ f2 = true; q2 = query(t->r, mid+2, hi); }

   if (!f1 && !f2){
      ans.best = ans.suf = ans.pref = ans.sum = t->key;
      return ans;
   }

   if (f1 && f2){
      ans.merge(&q1, t->key, &q2);
   }
   else if (f1){
      if (x <= mid+1 && mid+1 <= y) ans.merge(&q1, t->key, NULL);
      else ans = q1;
   }
   else if (x <= mid+1 && mid+1 <= y){
      ans.merge(NULL, t->key, &q2);
   }
   else{
      ans = q2;
   }
   return ans;
}

void merge(node_t &t, node_t* l, node_t* r){
   if (!l || !r)
      t = l ? l : r;
   else if (l->pr > r->pr)
      merge(l->r, l->r, r), t = l;
   else
      merge(r->l, l, r->l), t = r;
   fix(t);
}

void split (node_t* t, node_t* &l, node_t* &r, int pos, int add = 0){
   if (!t)
      return void (l = r = 0);

   int cur_pos = add + ((t->l)?(t->l->sz):(0));
   if (pos <= cur_pos)
      split(t->l, l, t->l, key, add), r = t;
   else
      split(t->r, t->r, r, key, cur_pos+1), l = t;
   fix(t);
}
```

## AVL Tree

```
struct Node{
   Node *l, *r; int h, size, key;
   Node(int k) : l(0), r(0), h(1), size(1), key(k) {}
   void u(){
      h = 1 + max(l ? l->h : 0, r ? r->h : 0);
      size = (l ? l->size : 0) + 1 + (r ? r->size : 0);
   }
};

Node *rotl(Node *x){ Node *y=x->r; x->r=y->l; y->l=x; x->u(); y->u(); return y; }
Node *rotr(Node *x){ Node *y=x->l; x->l=y->r; y->r=x; x->u(); y->u(); return y; }

Node *rebalance(Node *x) {
   x->u();
   if (x->l->h > 1 + x->r->h){
      if (x->l->l->h < x->l->r->h) x->l = rotl(x->l);
      x = rotr(x);
   }
   else if (x->r->h > 1 + x->l->h){
      if (x->r->r->h < x->r->l->h) x->r = rotr(x->r);
      x = rotl(x);
   }
   return x;
}

Node *insert(Node *x, int key){
   if (x == NULL) return new Node(key);
   if (key < x->key) x->l = insert(x->l, key); else x->r = insert(x->r, key);
   return rebalance(x);
}
```

## Find index with given cumulative frequency

```
// if in tree exists more than one index with a same
// cumulative frequency, this procedure will return
// some of them (we do not know which one)

// bitMask - initialy, it is the greatest bit of MaxVal
// bitMask store interval which should be searched
int find(int cumFre){
   int idx = 0; // this var is result of function

   while ((bitMask != 0) && (idx < MaxVal)){ // nobody likes overflow :)
      int tIdx = idx + bitMask; // we make midpoint of interval
      if (cumFre == tree[tIdx]) // if it is equal, we just return idx
         return tIdx;
      else if (cumFre > tree[tIdx]){
            // if tree frequency "can fit" into cumFre,
            // then include it
         idx = tIdx; // update index
         cumFre -= tree[tIdx]; // set frequency for next loop
      }
      bitMask >>= 1; // half current interval
   }
   if (cumFre != 0) // maybe given cumulative frequency doesn't exist
      return -1;
   else
      return idx;
}

// if in tree exists more than one index with a same
// cumulative frequency, this procedure will return
// the greatest one
int findG(int cumFre){
   int idx = 0;

   while ((bitMask != 0) && (idx < MaxVal)){
      int tIdx = idx + bitMask;
      if (cumFre >= tree[tIdx]){
         // if current cumulative frequency is equal to cumFre,
         // we are still looking for higher index (if exists)
         idx = tIdx;
         cumFre -= tree[tIdx];
      }
      bitMask >>= 1;
   }
   if (cumFre != 0)
      return -1;
   else
      return idx;
}
```

## Heap

```
struct heap{
   int heap[MAXV][2], v2n[MAXV];
   int size;

   void init(int sz) __attribute__((always_inline)){
      memset(v2n, -1, sizeof(int) * sz);
      size = 0;
   }

   void swap(int& a, int& b) __attribute__((always_inline)){
      int temp = a;
      a = b;
      b = temp;
   }

   void s(int a, int b) __attribute__((always_inline)){
      swap(v2n[heap[a][1]], v2n[heap[b][1]]);
      swap(heap[a][0], heap[b][0]);
      swap(heap[a][1], heap[b][1]);
   }

   int extract_min(){
      int ret = heap[0][1];
      s(0, --size);

      int cur = 0, next = 2;
      while (next < size){
         if (heap[next][0] > heap[next - 1][0])
            next--;
         if (heap[next][0] >= heap[cur][0])
            break;

         s(next, cur);
         cur = next;
         next = 2*cur + 2;
      }
      if (next == size && heap[next - 1][0] < heap[cur][0])
         s(next - 1, cur);
```

```
        return ret;
    }

    void decrease_key(int vertex, int new_value) __attribute__((always_inline)){
        if (v2n[vertex] == -1){
            v2n[vertex] = size;
            heap[size++][1] = vertex;
        }

        heap[v2n[vertex]][0] = new_value;
        int cur = v2n[vertex];
        while (cur >= 1){
            int parent = (cur - 1)/2;
            if (new_value >= heap[parent][0])
                break;

            s(cur, parent);
            cur = parent;
        }
    }
};
```

## Big Integer

```
const int DIG = 4;
const int BASE = 10000; // BASE**3 < 2**51
const int TAM = 2048;

struct bigint{
    int v[TAM], n;
    bigint(int x = 0): n(1){
        memset(v, 0, sizeof(v));
        v[n++] = x; fix();
    }
    bigint(char *s): n(1){
        memset(v, 0, sizeof(v));
        int sign = 1;
        while (*s && !isdigit(*s)) if (*s++ == '-') sign *= -1;
        char *t = strdup(s), *p = t + strlen(t);
        while (p > t){
            *p = 0; p = max(t, p - DIG);
            sscanf(p, "%d", &v[n]);
            v[n++] *= sign;
        }
        free(t); fix();
    }
    bigint& fix(int m = 0){
        n = max(m, n);
        int sign = 0;
        for (int i = 1, e = 0; i <= n || e && (n = i); i++){
            v[i] += e; e = v[i] / BASE; v[i] %= BASE;
            if (v[i]) sign = (v[i] > 0) ? 1 : -1;
        }

        for (int i = n - 1; i > 0; i--)
            if (v[i] * sign < 0){ v[i] += sign * BASE; v[i+1] -= sign; }
        while (n && !v[n]) n--;
        return *this;
    }

    int cmp(const bigint& x = 0) const {
        int i = max(n, x.n), t = 0;
        while (1) if ((t = ::cmp(v[i], x.v[i])) || i-- == 0) return t;
    }
    bool operator <(const bigint& x) const { return cmp(x) < 0; }
    bool operator ==(const bigint& x) const { return cmp(x) == 0; }
    bool operator !=(const bigint& x) const { return cmp(x) != 0; }

    operator string() const {
        ostringstream s; s << v[n];
        for (int i = n - 1; i > 0; i--) {
            s.width(DIG); s.fill('0'); s << abs(v[i]);
        }
        return s.str();
    }
    friend ostream& operator <<(ostream& o, const bigint& x){
        return o << (string) x;
    }

    bigint& operator +=(const bigint& x){
        for (int i = 1; i <= x.n; i++) v[i] += x.v[i];
        return fix(x.n);
    }
    bigint operator +(const bigint& x){ return bigint(*this) += x; }
    bigint& operator -=(const bigint& x){
        for (int i = 1; i <= x.n; i++) v[i] -= x.v[i];
        return fix(x.n);
    }
    bigint operator -(const bigint& x){ return bigint(*this) -= x; }
    bigint operator -(){ bigint r = 0; return r -= *this; }
    void ams(const bigint& x, int m, int b){ // *this += (x * m) << b;
        for (int i = 1, e = 0; (i <= x.n || e) && (n = i + b); i++){
            v[i+b] += x.v[i] * m + e; e = v[i+b] / BASE; v[i+b] %= BASE;
        }
```

```
    }
    bigint operator *(const bigint& x) const {
        bigint r;
        for (int i = 1; i <= n; i++) r.ams(x, v[i], i-1);
        return r;
    }
    bigint& operator *=(const bigint& x){ return *this = *this * x; }
    // cmp(x / y) == cmp(x) * cmp(y); cmp(x % y) == cmp(x);
    bigint div(const bigint& x){
        if (x == 0) return 0;
        bigint q; q.n = max(n - x.n + 1, 0);
        int d = x.v[x.n] * BASE + x.v[x.n-1];
        for (int i = q.n; i > 0; i--){
            int j = x.n + i - 1;
            q.v[i] = int((v[j] * double(BASE) + v[j-1]) / d);
            ams(x, -q.v[i], i-1);
            if (i == 1 || j == 1) break;
            v[j-1] += BASE * v[j]; v[j] = 0;
        }
        fix(x.n); return q.fix();
    }
    bigint& operator /=(const bigint& x){ return *this = div(x); }
    bigint& operator %=(const bigint& x){ div(x); return *this; }
    bigint operator /(const bigint& x){ return bigint(*this).div(x); }
    bigint operator %(const bigint& x){ return bigint(*this) %= x; }
    bigint pow(int x){
        if (x < 0) return (*this == 1 || *this == -1) ? pow(-x) : 0;
        bigint r = 1;
        for (int i = 0; i < x; i++) r *= *this;
        return r;
    }
    bigint root(int x){
        if (cmp() == 0 || cmp() < 0 && x % 2 == 0) return 0;
        if (*this == 1 || x == 1) return *this;
        if (cmp() < 0) return -(-*this).root(x);
        bigint a = 1, d = *this;

        while (d != 1){
            bigint b = a + (d /= 2);
            if (cmp(b.pow(x)) >= 0) { d += 1; a = b; }
        }
        return a;
    }
};
```

# Games

## Grundy numbers

For a two-player, normal-play (last to move wins) game on a graph $(V, E)$: $G(x) = mex(\{G(y) : (x, y) \in E\})$, where $mex(S) = min\{n \geq 0 : n \notin S\}$. $x$ is losing iff $G(x) = 0$.

## Sums of games

- Player chooses a game and makes a move in it. Grundy number of a position is xor of grundy numbers of positions in summed games.
- Player chooses a non-empty subset of games (possibly, all) and makes moves in all of them. A position is losing iff each game is in a losing position.
- Player chooses a proper subset of games (not empty and not all), and makes moves in all chosen ones. A position is losing iff grundy numbers of all games are equal.
- Player must move in all games, and loses if can't move in some game. A position is losing if any of the games is in a losing position.

## Misère Nim

A position with pile sizes $a_1, a_2, ..., a_n \geq 1$, not all equal to 1, is losing iff $a_1 \bigoplus a_2 \bigoplus ... \bigoplus a_n = 0$ (like in normal nim.) A position with $n$ piles of size 1 is losing iff $n$ is odd.

# Math

## Polynomials

```
typedef complex<double> cdouble;

const double eps = 1e-9;
const double inf = 1e50;

int cmp(double a, double b){
   if(a - b > eps) return 1;
   if(b - a > eps) return -1;
   return 0;
}

int cmp(cdouble x, cdouble y = 0){
   return cmp(abs(x), abs(y));
}

const int MAX = 200;

struct poly{
   vector<cdouble> p;
   int n;

   poly(int n = 0) : n(n), p(vector<cdouble>(MAX)) {}
   poly(vector<cdouble> v) : n(v.size()), p(v) {}

   cdouble& operator [](int i){ return p[i]; }

   //Calcula a derivada de P(x)
   poly derivate(){
      poly r(n-1);
      FOR(i, 1, n) {
         r[i-1] = p[i] * cdouble(i);
      }
      return r;
   }

   //Divides P(x) by (x - z)
   //Returns in the form Q(x) + r
   pair<poly, cdouble> ruffini(cdouble z){
      if (n == 0) return MP(poly(), 0);
      poly r(n - 1);

      RFOR(i, n, 1){
         r[i - 1] = r[i] * z + p[i];
      }

      return mp(r, r[0] * z + p[0]);
   }

   //Return P(x) mod (x - z)
   cdouble operator % (cdouble z){
      return ruffini(z).second;
   }

   cdouble find_one_root(cdouble x){
      poly p0 = *this;
      poly p1 = p0.derivate();
      poly p2 = p1.derivate();

      int m = 1000; //gives 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.PB(z);
      }
      return r;
   }
};
```

## Simpson's rule

```
double f(double x){
   //use function here
}

//Integra f(x) no intervalo [a, b] em O(k)
double simpson(double a, double b, int k = 1000){
   double dx, x, t = 0.0;
   dx = (b - a) / (2.0 * k);
   REP(i, k){
      t += (i == 0 ? 1.0 : 2.0) * f(a + 2.0 * i * dx);
      t += 4.0 * f(a + (2.0 * i + 10) * dx);
   }
   t += f(b);

   return t * (b - a) / (6.0 * k);
}
```

## Cycle Finding

```
//Brend cycle finding algorithm
//Retorna o tamanho do ciclo

int f(int x){
   //Returns next sequence term
}

int cycle_find(int x0){
   int pow = 1, len = 1;
   int t = x0, h = f(x0);
   while (t != h){
      if (len == pow){
         t = h;
         pow <<= 1;
         len = 0;
      }
      h = f(h);
      len++;
   }
   return len;
}
```

## Romberg's Method

```
long double romberg(long double a, long double b,
                long double(*func)(long double)){
   long double approx[2][25];
   long double *cur = approx[1], *prev = approx[0];

   prev[0] = 1/2.0 * (b-a) * (func(a) + func(b));
   for(int it = 1; it < 25; ++it, swap(cur, prev)) {
      if(it > 1 && cmp(prev[it-1], prev[it-2]) == 0)
         return prev[it-1];

      cur[0] = 1/2.0 * prev[0];
      long double div = (b-a)/pow(2, it);
      for (long double sample = a + div; sample < b; sample += 2 * div)
         cur[0] += div * func(a + sample);

      for (int j = 1; j <= it; ++j)
         cur[j] = cur[j-1] + 1/(pow(4, it) - 1)*(cur[j-1] + prev[j-1]);
   }

   return prev[24];
}
```

## FFT

```
typedef complex<long double> Complex;
long double PI = 2 * acos(0.0L);
// Decimation-in-time radix-2 FFT.
//
// Computes in-place the following transform:
// y[i] = A(w^(dir*i)),
// where
// w = exp(2pi/N) is N-th complex principal root of unity,
// A(x) = a[0] + a[1] x + ... + a[n-1] x^{n-1}m
// dir \in {-1, 1} is FFTs direction (+1=forward, -1=inverse).
//
// Notes:
// * N must be a power of 2,
// * scaling by 1/N after inverse FFT is callers resposibility.
void FFT(Complex *a, int N, int dir){
   int lgN;
   for (lgN = 1; (1 << lgN) < N; lgN++);
   assert((1 << lgN) == N);

   for (int i = 0; i < N; ++i){
      int j = 0;
      for (int k = 0; k < lgN; ++k)
         j |= ((i>>k)&1) << (lgN-1-k);
      if (i < j) swap(a[i], a[j]);
   }
   for (int s = 1; s <= lgN; ++s){
      int h = 1 << (s - 1);
      Complex t, w, w_m = exp(Complex(0, dir*PI/h));
      for (int k = 0; k < N; k += h+h){
         w = 1;
         for (int j = 0; j < h; ++j){
            t = w * a[k+j+h];
            a[k+j+h] = a[k+j] - t;
            a[k+j] += t;
            w *= w_m;
         }
      }
   }
}
```

## Simplex

```
const double EPS = 1e-9;
typedef long double T;
typedef vector<T> VT;
vector<VT> A;
VT b, c, res;
VI kt, N;
int m;

void pivot(int k, int l, int e){
   int x = kt[l]; T p = A[l][e];
   REP(i, k) A[l][i] /= p; b[l] /= p; N[e] = 0;
   REP(i, m) if (i!=l) b[i] -= A[i][e]*b[l], A[i][x] = A[i][e]*-A[l][x];
   REP(j, k) if (N[j]){
      c[j] -= c[e]*A[l][j];
      REP(i, m) if (i!=l) A[i][j] -= A[i][e]*A[l][j];
   }
   kt[l] = e; N[x] = 1; c[x] = c[e]*-A[l][x];
}

VT doit(int k){
   VT res; T best;
   while (1){
      int e = -1, l = -1;
      REP(i, k) if (N[i] && c[i] > EPS){ e = i; break; }
      if (e == -1) break;
      REP(i, m) if (A[i][e]>EPS && (l == -1 || best > b[i]/A[i][e]))
         best = b[ l=i ]/A[i][e];
      if (l == -1) /*ilimitado*/ return VT();
      pivot(k, l, e);
   }
   res.resize(k, 0); REP(i, m) res[kt[i]] = b[i];
   return res;
}

VT simplex(vector<VT> &AA, VT &bb, VT &cc){
   int n = AA[0].size(), k; m = AA.size(); k = n+m+1;
   kt.resize(m); b = bb; c = cc; c.resize(n+m); A = AA;
   REP(i, m){ A[i].resize(k); A[i][n+i] = 1; A[i][k-1] = -1; kt[i] = n+i; }
   N = VI(k, 1); REP(i,m) N[kt[i]] = 0;
   int pos = min_element(ALL(b))-b.begin();
   if (b[pos] < -EPS){
      c = VT(k, 0); c[k-1] = -1; pivot(k, pos, k-1); res = doit(k);
      if (res[k-1] > EPS) /*impossivel*/ return VT();
      REP(i, m) if (kt[i] == k-1)
         REP(j, k-1) if (N[j] && (A[i][j] < -EPS || EPS < A[i][j])){
            pivot(k, i, j); break;
         }
      c = cc; c.resize(k, 0); REP(i, m) REP(j, k) if (N[j]) c[j] -= c[kt[i]]*A[i][j];
   }
   res = doit(k-1); if (!res.empty()) res.resize(n);
   return res;
}
```

## Matrix Exp

```
#define MAXN 111

int n;

void mult(double x[][MAXN], double y[][MAXN]) {
   double aux[n][n];
   REP(i, n) REP(j, n) aux[i][j] = 0.0;

   REP(i, n) REP(j, n) REP(k, n){
      aux[i][j] += (x[i][k] * y[k][j]);
   }

   REP(i, n) REP(j, n) x[i][j] = aux[i][j];
}

void powM(double mat[][MAXN], int ex) {
   //Inicializa com a identidade
   double res[MAXN][MAXN];
   REP(i, n) REP(j, n) res[i][j] = (i == j) ? 1.0 : 0.0;
   while(ex) {
      if(ex & 1) mult(res, mat);
      mult(mat, mat);
      ex >>= 1;
   }

   REP(i, n) REP(j, n) mat[i][j] = res[i][j];
}
```

# Geometry

## Pick's theorem

$I = A - B/2 + 1$, where $A$ is the area of a lattice polygon, $I$ is the number of lattice points inside it, and $B$ is the number of lattice points on the boundary. Number of lattice points minus one on a line segment from (0, 0) and (x, y) is gcd(x, y).

$a.b = a_x b_x + a_y b_y = |a|.|b|.cos(\theta)$
$a \times b = a_x b_y - a_y b_x = |a|.|b|.sin(\theta)$
3D: $a \times b = (a_y b_z - a_z b_y, a_z b_x - a_x b_z, a_x b_y - a_y b_x)$

## Line

Line $ax + by = c$ through $A(x_1, y_1)$ and $B(x_2, y_2)$: $a = y_1 - y_2$, $c = x_2 - x_1$, $c = ax_1 + by_1$.
Half-plane to the left of the directed segment $AB$: $ax + by \geq c$.
Normal vector: $(a, b)$. Direction vector: $(b, -a)$. Perpendicular line: $-bx + ay = d$.
Point of intersection of $a_1 x + b_1 y = c_1$ and $a_2 x + b_2 y = c_2$ is $\frac{1}{a_1 b_2 - a_2 b_1}(c_1 b_2 - c_2 b_1, a_1 c_2 - a_2 c_1)$.
Distance from line $ax + by + c = 0$ to point $(x_0, y_0)$ is $|ax_0 + by_0 + c|/\sqrt{a^2 + b^2}$.
Distance from line AB to P (for any dimension): $\frac{|(A-P)\times(B-P)|}{|A-B|}$.
Point-line segment distance:
if $(dot(B - A, P - A) < 0)$ return $dist(A, P)$;
if $(dot(A - B, P - B) < 0)$ return $dist(B, P)$;
return $fabs(cross(P, A, B)/dist(A, B))$;

## Projection

Projection of point $C$ onto line $AB$ is $\frac{AB.AC}{AB.AB}AB$.
Projection of $(x_0, y_0)$ onto line $ax + by = c$ is $(x_0, y_0) + \frac{1}{a^2+b^2}(ad, bd)$, where $d = c - ax_0 - by_0$.
Projection of the origin is $\frac{1}{a^2+b^2}(ac, bc)$.

## Segment-segment intersection

Two line segments intersect if one of them contains an endpoint of the other segment, or each segment straddles the line, containing the other segment ($AB$ straddles line $l$ if $A$ and $B$ are on the opposite sides of $l$.)

### Circle-circle and circle-line intersection

```
a = x2 - x1;   b = y2 - y1;   c = [(r1^2 - x1^2 - y1^2) - (r2^2 - x2^2 - y2^2)] / 2;
d = sqrt(a^2 + b^2);
if not |r1 - r2| <= d <= |r1 + r2|, return "no solution"
if d == 0, circles are concentric, a special case
// Now intersecting circle (x1,y1,r1) with line ax+by=c
Normalize line: a /= d; b /= d; c /= d;     // d=sqrt(a^2+b^2)
e = c - a*x1 - b*y1;
h = sqrt(r1^2 - e^2);                       // check if r1<e for circle-line test
return (x1, y1) + (a*e, b*e) +/- h*(-b, a);
```

### Circle from 3 points (circumcircle)

Intersect two perpendicular bisectors. Line perpendicular to $ax + by = c$ has the form $-bx + ay = d$. Find $d$ by substituting midpoint's coordinates.

### Angular bisector

Angular bisector of angle $ABC$ is line $BD$, where $D = \frac{BA}{|BA|} + \frac{BC}{|BC|}$.
Center of incircle of triangle $ABC$ is at the intersection of angular bisectors, and is $\frac{a}{a+b+c}A + \frac{b}{a+b+c}B + \frac{c}{a+b+c}C$ where $a$, $b$, $c$ are lengths of sides, opposite to vertices $A$, $B$, $C$. Radius $= \frac{2S}{a+b+c}$

### Counter-clockwise rotation around the origin

$(x, y) \to (x \cos\phi - y \sin\phi, x \sin\phi + y \cos\phi)$.
90-degrees counter-clockwise rotation: $(x, y) \to (-y, x)$. Clockwise: $(x, y) \to (y, -x)$.

### 3D rotation

3D rotation by ccw angle $\phi$ around axis $n$: $r' = r\cos\phi + n(n \cdot r)(1 - \cos\phi) + (n \times r)\sin\phi$

### Plane equation from 3 points

$N \cdot (x, y, z) = N \cdot A$, where $N$ is normal: $N = (B - A) \times (C - A)$.

### 3D figures

Sphere: Volume $V = \frac{4}{3}\pi r^3$, surface area $S = 4\pi r^2$
$x = \rho \sin\theta \cos\phi$, $y = \rho \sin\theta \sin\phi$, $z = \rho \cos\theta$, $\phi \in [-\pi, \pi], \theta \in [0, \pi]$
Spherical section: Volume $V = \pi h^2(r - h/3)$, surface area $S = 2\pi r h$
Pyramid: Volume $V = \frac{1}{3} h S_{base}$
Cone: Volume $V = \frac{1}{3}\pi r^2 h$, lateral surface area $S = \pi r \sqrt{r^2 + h^2}$

### Area of a simple polygon

$\frac{1}{2}\sum_{i=0}^{n-1}(x_i y_{i+1} - x_{i+1} y_i)$, where $x_n = x_0$, $y_n = y_0$. Area is negative if the boundary is oriented clockwise.

### Winding number

Shoot a ray from given point in an arbitrary direction. For each intersection of ray with polygon's side, add +1 if the side crosses it counterclockwise, and -1 if clockwise.

### Range Tree

```
vector<pt> pts, tree[MAXSZ];
vector<TYPE> xs;
vector<int> lnk[MAXSZ][2];

int rt_recurse(int root, int left, int right) {
   lnk[root][0].clear(); lnk[root][1].clear(); tree[root].clear();

   if(left == right) {
      vector<pt>::iterator it;
      it = lower_bound(pts.begin(), pts.end(), pt(xs[left], -INF));
      for(; it != pts.end() && cmp(it->x, xs[left]) == 0; ++it)
         tree[root].push_back(*it);
      return tree[root].size();
   }

   int mid = (left + right)/2, cl = 2*root + 1, cr = cl + 1;
   int sz1 = rt_recurse(cl, left, mid);
   int sz2 = rt_recurse(cr, mid + 1, right);

   lnk[root][0].reserve(sz1+sz2+1);
   lnk[root][1].reserve(sz1+sz2+1);
   tree[root].reserve(sz1+sz2);

   int l = 0, r = 0, llink = 0, rlink = 0; pt last;
   while(l < sz1 || r < sz2) {
      if(r == sz2 || (l < sz1 && compy(tree[cl][l], tree[cr][r])))
         tree[root].push_back(last = tree[cl][l++]);
      else tree[root].push_back(last = tree[cr][r++]);

      while(llink < sz1 && compy(tree[cl][llink], last))
         ++llink;
      while(rlink < sz2 && compy(tree[cr][rlink], last))
         ++rlink;

      lnk[root][0].push_back(llink);
      lnk[root][1].push_back(rlink);
   }

   lnk[root][0].push_back(tree[cl].size());
   lnk[root][1].push_back(tree[cr].size());

   return tree[root].size();
}

void rt_build() {
   sort(pts.begin(), pts.end());
   xs.clear();
   for(int i = 0; i < pts.size(); ++i) xs.push_back(pts[i].x);
```

```
   xs.erase(unique(xs.begin(), xs.end()), xs.end());
   rt_recurse(0, 0, xs.size() - 1);
}

int rt_query(int root, int l, int r, TYPE a, TYPE b, TYPE c, TYPE d,
         int posl = -1, int posr = -1) {
   if(root == 0 && posl == -1) {
      posl = lower_bound(tree[0].begin(), tree[0].end(), pt(a, c), compy)
         - tree[0].begin();
      posr = upper_bound(tree[0].begin(), tree[0].end(), pt(b, d), compy)
         - tree[0].begin();
   }
   if(posl == posr) return 0;
   if(a <= xs[l] && xs[r] <= b)
      return posr - posl;

   int mid = (l+r)/2, ret = 0;
   if(cmp(a, xs[mid]) <= 0)
      ret += rt_query(2*root+1, l, mid, a, b, c, d,
               lnk[root][0][posl], lnk[root][0][posr]);
   if(cmp(xs[mid+1], b) <= 0)
      ret += rt_query(2*root+2, mid+1, r, a, b, c, d,
               lnk[root][1][posl], lnk[root][1][posr]);
   return ret;
}
```

## KD Tree

```
int tree[4*MAXSZ], val[4*MAXSZ];
TYPE split[4*MAXSZ];
vector<pt> pts;

void kd_recurse(int root, int left, int right, bool x){
   if (left == right){
      tree[root] = left;
      val[root] = 1;
      return;
   }

   int mid = (right+left)/2;
   nth_element(pts.begin() + left, pts.begin() + mid,
         pts.begin() + right + 1, x ? compx : compy);
   split[root] = x ? pts[mid].x : pts[mid].y;

   kd_recurse(2*root+1, left, mid, !x);
   kd_recurse(2*root+2, mid+1, right, !x);

   val[root] = val[2*root+1] + val[2*root+2];
}

void kd_build(){
   memset(tree, -1, sizeof tree);
   kd_recurse(0, 0, pts.size() - 1, true);
}

int kd_query(int root, TYPE a, TYPE b, TYPE c, TYPE d, TYPE ca = -INF,
         TYPE cb = INF, TYPE cc = -INF, TYPE cd = INF, bool x = true){
   if (a <= ca && cb <= b && c <= cc && cd <= d)
      return val[root];

   if (tree[root] != -1)
      return a <= pts[tree[root]].x && pts[tree[root]].x <= b &&
         c <= pts[tree[root]].y && pts[tree[root]].y <= d ? val[root] : 0;

   int ret = 0;
   if (x){
      if (a <= split[root])
         ret += kd_query(2*root+1, a, b, c, d, ca, split[root], cc, cd, !x);
      if (split[root] <= b)
         ret += kd_query(2*root+2, a, b, c, d, split[root], cb, cc, cd, !x);
   }
   else{
      if (c <= split[root])
         ret += kd_query(2*root+1, a, b, c, d, ca, cb, cc, split[root], !x);
      if (split[root] <= d)
         ret += kd_query(2*root+2, a, b, c, d, ca, cb, split[root], cd, !x);
   }
   return ret;
}

pt kd_neighbor(int root, pt a, bool x){
   if (tree[root] != -1)
      return a == pts[tree[root]] ? pt(INF, INF) : pts[tree[root]];

   TYPE num = x ? a.x : a.y;
   int term = num <= split[root] ? 1 : 2;
   pt ret;

   TYPE d = norm(a - (ret = kd_neighbor(2*root + term, a, !x)));
   if ((split[root] - num)*(split[root] - num) < d){
      pt ret2 = kd_neighbor(2*root + 3 - term, a, !x);
      if (norm(a - ret2) < d)
         ret = ret2;
   }

   return ret;
}
```

## Enclosing Circle

```
circle enclosing_circle(vector<pt>& pts){
   srand(unsigned(time(0)));
   random_shuffle(pts.begin(), pts.end());

   circle c(pt(), -1);
   for (int i = 0; i < pts.size(); ++i){
      if (point_circle(pts[i], c)) continue;
      c = circle(pts[i], 0);
      for (int j = 0; j < i; ++j){
         if (point_circle(pts[j], c)) continue;
         c = circle((pts[i] + pts[j])/2, abs(pts[i] - pts[j])/2);
         for (int k = 0; k < j; ++k){
            if (point_circle(pts[k], c)) continue;
            pt center = circumcenter(pts[i], pts[j], pts[k]);
            c = circle(center, abs(center - pts[i])/2);
         }
      }
   }
   return c;
}
```

## Line Sweep Applications

```
// using line sweep to get the minimum distance between a pair of points
template<typename T> T inline SQR( const T &a ){ return a*a; }

double min_dist(vector< pair<double,double> > &point){
   // Ordena os pontos pelo X que vai ser o eixo percorrido no line sweep
   sort(point, point+N);
   // Comeca a menor distancia com um valor grande o suficiente
   double h = 1e10;
   // Conjunto de pontos ativos, usa uma funcao que ordena eles pelo y
   set< pair<double,double>, comp > active;
   int pactive = 0;
   for (int i = 0; i < N; ++i){ // Comeca a varrer o eixo X

      // Tira os pontos que estavam no conjuntos de ativos e nao tem mais
      // chance de melhorarem a menor distancia
      while (pactive < i && point[pactive].first < point[i].first-h){
         active.erase( point[pactive] );
         pactive++;
      }

      // Limita os pontos a serem verificados numa box de interesse
      set< pair<double,double> >:: iterator lb,ub;
      lb = active.lower_bound( make_pair( -1000000, point[i].second-h) );
      ub = active.upper_bound( make_pair( +1000000, point[i].second+h) );

      // Verifica se algum dos pontos na box melhora a distancia
      while (lb != ub){
         double hh = sqrt(SQR(point[i].second-(lb->second))
                    + SQR(point[i].first-(lb->first)));
         if (hh < h) h = hh;
         lb++;
      }
      // Adiciona o ponto atual no conjunto de ativos
      active.insert(point[i]);
   }
   return h;
}

// using line sweep to get the area of the union of rectangles O(n^2)
struct event_t{
   int ax, frm;
   char wut;
   event_t(int a = 0, int b = 0, char c = 0) : ax(a), frm(b), wut(c) {}
   bool operator < (const event_t& a) const {
      if (ax != a.ax) return ax < a.ax;
      return wut < a.wut;
   }
};

struct rect_t{
   int x1, x2, y1, y2;
   rect_t(int a = 0, int b = 0, int c = 0, int d = 0)
      : x1(a), y1(b), x2(c), y2(d) {}
};

int area(vector<rect_t> rect) {
   vector< event_t > eventx, eventy;
   for (size_t i = 0, sz = rect.size(); i < sz; ++i){
      eventx.pb(event_t(rect[i].x1, i, 0));
      eventx.pb(event_t(rect[i].x2, i, 1));
      eventy.pb(event_t(rect[i].y1, i, 0));
      eventy.pb(event_t(rect[i].y2, i, 1));
   }

   sort(eventx.begin(), eventx.end());
   sort(eventy.begin(), eventy.end());
```

```
   vector< bool > active(int(rect.size()), false);

   active[eventx[0].frm] = true;

   int ans = 0;
   for (size_t i = 1, sz = eventx.size(); i < sz; ++i){

      int in = 0;
      int lst = 0;
      for (size_t j = 0, szz = eventy.size(); j < szz; ++j){
         if (!active[eventy[j].frm]) continue;
         if (in){
            ans += (eventy[j].ax-lst)*(eventx[i].ax-eventx[i-1].ax);
            lst = eventy[j].ax;
         }
         else lst = eventy[j].ax;

         if (eventy[j].wut) in--;
         else in++;
      }

      if (eventx[i].wut) active[eventx[i].frm] = false;
      else active[eventx[i].frm] = true;
   }

   return ans;
}
```

## Closest Points

```
pair<pt, pt> closest_points_rec(vector<pt>& px, vector<pt>& py) {
   pair<pt, pt> ret;
   double d;

   if(px.size() <= 3) {
      double best = 1e10;
      for(int i = 0; i < px.size(); ++i)
         for(int j = i + 1; j < px.size(); ++j)
            if(dist(px[i], px[j]) < best) {
               ret = make_pair(px[i], px[j]);
               best = dist(px[i], px[j]);
            }

      return ret;
   }

   pt split = px[(px.size() - 1)/2];
   vector<pt> qx, qy, rx, ry;
   for(int i = 0; i < px.size(); ++i)
      if(px[i] <= split) qx.push_back(px[i]);
      else rx.push_back(px[i]);

   for(int i = 0; i < py.size(); ++i)
      if(py[i] <= split) qy.push_back(py[i]);
      else ry.push_back(py[i]);

   ret = closest_points_rec(qx, qy);
   pair<pt, pt> rans = closest_points_rec(rx, ry);
   double delta = dist(ret.first, ret.second);

   if((d = dist(rans.first, rans.second)) < delta) {
      delta = d;
      ret = rans;
   }

   vector<pt> s;
   for(int i = 0; i < py.size(); ++i)
      if(cmp(abs(py[i].x - split.x), delta) <= 0)
         s.push_back(py[i]);

   for(int i = 0; i < s.size(); ++i)
      for(int j = 1; j <= 7 && i + j < s.size(); ++j)
         if((d = dist(s[i], s[i+j])) < delta) {
            delta = d;
            ret = make_pair(s[i], s[i+j]);
         }

   return ret;
}

pair<pt, pt> closest_points(vector<pt> pts) {
   if(pts.size() == 1) return make_pair(pt(-INF, -INF), pt(INF, INF));

   sort(pts.begin(), pts.end());
   for(int i = 0; i + 1 < pts.size(); ++i)
      if(pts[i] == pts[i+1])
         return make_pair(pts[i], pts[i+1]);

   vector<pt> py = pts;
   sort(py.begin(), py.end(), compy);
   return closest_points_rec(pts, py);
}
```

## ISECT Primitives

```
bool in_rect(pt a, pt b, pt c) {
   return sgn(c.x - min(a.x, b.x)) >= 0 && sgn(max(a.x, b.x) - c.x) >= 0 &&
      sgn(c.y - min(a.y, b.y)) >= 0 && sgn(max(a.y, b.y) - c.y) >= 0;
}
bool ps_isects(pt a, pt b, pt c) { return ccw(a,b,c) == 0 && in_rect(a,b,c); }

bool ss_isects(pt a, pt b, pt c, pt d) {
   if (ccw(a,b,c)*ccw(a,b,d) == -1 && ccw(c,d,a)*ccw(c,d,b) == -1) return true;
   return ps_isects(a, b, c) || ps_isects(a, b, d) ||
      ps_isects(c, d, a) || ps_isects(c, d, b);
}
```

```
pt parametric_isect(pt p, pt v, pt q, pt w) {
   double t = ((q-p)%w)/(v%w);
   return p + v*t;
}

pt ss_isect(pt p, pt q, pt r, pt s) {
   pt isect = parametric_isect(p, q-p, r, s-r);
   if(ps_isects(p, q, isect) && ps_isects(r, s, isect)) return isect;
   return pt(1/0.0, 1/0.0);
}
```

## Misc Primitives

```
bool point_circle(pt p, circle c) {
   return cmp(abs(p - c.c), c.r) <= 0;
}

double ps_distance(pt p, pt a, pt b) {
   p = p - a; b = b - a;
   double coef = min(max((b||p)/(b||b), TYPE(0)), TYPE(1));
   return abs(p - b*coef);
}
```

```
pt circumcenter(pt a, pt b, pt c) {
   return parametric_isect((b+a)/2, (b-a)*I, (c+a)/2, (c-a)*I);
}

bool compy(pt a, pt b) {
   return cmp(a.y, b.y) ? cmp(a.y, b.y) < 0 : cmp(a.x, b.x) < 0;
}
bool compx(pt a, pt b) { return a < b; }
```

## Point

```
typedef double TYPE;
const TYPE EPS = 1e-9, INF = 1e9;

inline int sgn(TYPE a) { return a > EPS ? 1 : (a < -EPS ? -1 : 0); }
inline int cmp(TYPE a, TYPE b) { return sgn(a - b); }

struct pt {
   TYPE x, y;
   pt(TYPE x = 0, TYPE y = 0) : x(x), y(y) { }

   bool operator==(pt p) { return cmp(x, p.x) == 0 && cmp(y, p.y) == 0; }
   bool operator<(pt p) const {
      return cmp(x, p.x) ? cmp(x, p.x) < 0 : cmp(y, p.y) < 0;
   }
   bool operator<=(pt p) { return *this < p || *this == p; }
   TYPE operator||(pt p) { return x*p.x + y*p.y; }
   TYPE operator%(pt p) { return x*p.y - y*p.x; }
   pt operator~() { return pt(x, -y); }
   pt operator+(pt p) { return pt(x + p.x, y + p.y); }
   pt operator-(pt p) { return pt(x - p.x, y - p.y); }
   pt operator*(pt p) { return pt(x*p.x - y*p.y, x*p.y + y*p.x); }
   pt operator/(TYPE t) { return pt(x/t, y/t); }
   pt operator/(pt p) { return (*this * ~p)/(p||p); }
};
const pt I = pt(0,1);

struct circle {
   pt c; TYPE r;
   circle(pt c, TYPE r) : c(c), r(r) { }
};

TYPE norm(pt a) { return a||a; }
TYPE abs(pt a) { return sqrt(a||a); }
TYPE dist(pt a, pt b) { return abs(a - b); }
```

```
TYPE area(pt a, pt b, pt c) { return (a-c)%(b-c); }
int ccw(pt a, pt b, pt c) { return sgn(area(a, b, c)); }
pt unit(pt a) { return a/abs(a); }
double arg(pt a) { return atan2(a.y, a.x); }
pt f_polar(TYPE mod, double ang) { return pt(mod * cos(ang), mod * sin(ang)); }
inline int g_mod(int i, int n) { if(i == n) return 0; return i; }

ostream& operator<<(ostream& o, pt p) {
   return o << "(" << p.x << "," << p.y << ")";
}
```

## Polygon Primitives

```
double p_signedarea(vector<pt>& pol) {
   double ret = 0;
   for(int i = 0; i < pol.size(); ++i)
      ret += pol[i] % pol[g_mod(i+1, pol.size())];
   return ret/2;
}

int point_polygon(pt p, vector<pt>& pol) {
   int n = pol.size(), count = 0;

   for(int i = 0; i < n; ++i) {
      int i1 = g_mod(i+1, n);
      if (ps_isects(pol[i], pol[i1], p)) return -1;
      else if(((sgn(pol[i].y - p.y) == 1) != (sgn(pol[i1].y - p.y) == 1)) &&
            ccw(pol[i], p, pol[i1]) == sgn(pol[i].y - pol[i1].y)) ++count;
   }
   return count % 2;
}
```

## Convex Hull

```
pt pivot;

bool hull_comp(pt a, pt b) {
   int turn = ccw(a, b, pivot);
   return turn == 1 || (turn == 0 && cmp(norm(a-pivot), norm(b-pivot)) < 0);
}

vector<pt> hull(vector<pt> pts) {
   if(pts.size() <= 1) return pts;
   vector<pt> ret;

   int mini = 0;
   for(int i = 1; i < pts.size(); ++i)
      if(pts[i] < pts[mini])
         mini = i;

   pivot = pts[mini];
   swap(pts[0], pts[mini]);
   sort(pts.begin() + 1, pts.end(), hull_comp);

   ret.push_back(pts[0]);
   ret.push_back(pts[1]);
   int sz = 2;

   for(int i = 2; i < pts.size(); ++i) {
      while(sz >= 2 && ccw(ret[sz-2], ret[sz-1], pts[i]) <= 0)
         ret.pop_back(), --sz;
      ret.push_back(pts[i]), ++sz;
   }

   return ret;
}
```

# Dynamic Programming

## Longest Ascending Subsequence

```
/*
  Description: Given an array of size n, asc_seq returns the length
           of the longest ascending subsequence, as well as one
        of the subsequences in S.

      sasc_seq returns the length of the longest strictly
      ascending subsequence.

  Complexity: O(n log n)

  Notes: If you want to do the same things with descending
           subsequences, just reverse the array before calling
      the routines.
*/

#include <stdio.h>
#include <stdlib.h>
#include <assert.h>

int asc_seq(int *A, int n, int *S){
 int *m, *seq, i, k, low, up, mid, start;

 m = malloc((n+1) * sizeof(int));
 seq = malloc(n * sizeof(int));
 /* assert(m && seq); */

 for (i = 0; i < n; i++) seq[i] = -1;
 m[1] = start = 0;
 for (k = i = 1; i < n; i++) {
  if (A[i] >= A[m[k]]) {
   seq[i] = m[k++];
   start = m[k] = i;
  } else if (A[i] < A[m[1]]) {
   m[1] = i;
  } else {
   /* assert(A[m[1]] <= A[c] && A[c] < A[m[k]]); */
   low = 1;
   up = k;
   while (low != up-1) {
  mid = (low+up)/2;
  if(A[m[mid]] <= A[i]) low = mid;
  else up = mid;
   }
   seq[i] = m[low];
   m[up] = i;
  }
 }
 for (i = k-1; i >= 0; i--) {
  S[i] = A[start];
  start = seq[start];
 }
 free(m); free(seq);
 return k;
}

int sasc_seq(int *A, int n, int *S){
 int *m, *seq, i, k, low, up, mid, start;

 m = malloc((n+1) * sizeof(int));
 seq = malloc(n * sizeof(int));
 /* assert(m && seq); */

 for (i = 0; i < n; i++) seq[i] = -1;
 m[1] = start = 0;
 for (k = i = 1; i < n; i++) {
  if (A[i] > A[m[k]]) {
   seq[i] = m[k++];
   start = m[k] = i;
  } else if (A[i] < A[m[1]]) {
   m[1] = i;
  } else if (A[i] < A[m[k]]) {
   low = 1;
   up = k;
   while (low != up-1) {
  /* assert(A[m[h]] <= A[c] && A[c] < A[m[j]]); */
  mid = (low+up)/2;
  if(A[m[mid]] <= A[i]) low = mid;
  else up = mid;
   }
   if (A[i] > A[m[low]]) {
  seq[i] = m[low];
  m[up] = i;
   }
  }
 }
 for (i = k-1; i >= 0; i--) {
  S[i] = A[start];
  start = seq[start];
 }
 free(m); free(seq);
 return k;
}

int main(void)
```

```
{
 int *A, *S, n, i, k;

 while (scanf("%d", &n) == 1 && n > 0) {
  A = malloc(n*sizeof(int));
  S = malloc(n*sizeof(int));

  /* Read in array */
  for (i = 0; i < n; i++) scanf("%d", &A[i]);

  /* Find longest ascending subsequence */
  k = asc_seq(A, n, S);
  printf("length␣=␣%d\n", k);
  for (i = 0; i < k; i++){
   printf("%d␣", S[i]);
  }
  printf("\n");

  /* Find longest strictly ascending subsequence */
  k = sasc_seq(A, n, S);
  printf("length␣=␣%d\n", k);
  for (i = 0; i < k; i++) {
   printf("%d␣", S[i]);
  }
  printf("\n");
  free(A);
  free(S);
 }
 return 0;
}
```

## Integer Partitioning

```
/*
  Description: Template for calculating the number of ways of
           partitioning the integer N into M parts.

  Complexity: O(N^2)

  Notes: A partition of a number N is a representation of
           N as the sum of positive integers
      e.g. 5 = 1+1+1+1+1

      The number of ways of partitioning an integer N
      into M parts is equal to the number of ways of
      partitioning the number N with the largest element
      being of size M. This is best seen with a Ferres-
      Young diagram:
      Suppose N = 8, M = 3:

      4 = * * * *
            3 = * * *
            1 = *
         3 2 2 1
      By transposition from rows to columns, this equality
      can be seen.

      P(N, M) = P(N-1, M-1) + P(N-M, M)
      P(0, M) = P(N, 0) = 0
      P(N, 1) = 1
*/
```

```
#include <stdio.h>
#include <string.h>
#define MAXN 300
#define ULL unsigned long long

ULL A[MAXN+1][MAXN+1];

void Build(){
 int i, j;

 memset(A, 0, sizeof(A));
 A[0][0] = 1;
 for(i = 1; i <= MAXN; i++){
  A[i][1] = 1;
  for(j = 2; j <= i; j++)
   A[i][j] = A[i-1][j-1] + A[i-j][j];
 }
}

int main(){
 int n, m;

 Build();
 while(scanf("%d␣%d", &n, &m) == 2){
  printf("Partitions␣of␣%d␣into␣%d␣parts:␣%llu\n", n, m, A[n][m]);
 }
 return 0;
}
```

## Edit Distance (with path recovery)

```
/*
  Description: Computes the edit distance for two strings. Namely
           this involves certain costs for Replacement,
      Insertions and Deletions. Given these costs, and
      two words, this program calculates the minimum cost
      way to transform the first string to the second. An
      added bonus is the PathRecovery() subroutine, which
      prints out exactly what happens step by step.

  Complexity: O(N^2) where N is the size of the larger of the two
           strings

  Notes: Some modifications need to be made in order to fix
           path recovery.
*/

#include <stdio.h>
#include <string.h>

#define MAXN 90

char move[MAXN][MAXN]; /* Type of command used */
int g[MAXN][MAXN]; /* Cost of changes */

int editDistance(char *src, char *dst, int replace, int insert, int delete){
 int i, j, l1, l2;

 l1 = strlen(src);
 l2 = strlen(dst);

 for(j = 0; j <= l1; j++){
  g[0][j] = j;
  move[0][j] = 'D';
 }

 for(i = 1; i <= l2; i++){
  g[i][0] = i;
  move[i][0] = 'I';

  for(j = 1; j <= l1; j++){
   g[i][j] = g[i-1][j-1]+replace;
   move[i][j] = 'R';

   if(g[i-1][j]+insert < g[i][j]){
    g[i][j] = g[i-1][j]+insert;
    move[i][j] = 'I';
   }

   if(g[i][j-1]+delete < g[i][j]){
    g[i][j] = g[i][j-1]+delete;
    move[i][j] = 'D';
   }

   if(src[j-1] == dst[i-1] && g[i-1][j-1] < g[i][j]){
    g[i][j] = g[i-1][j-1];
    move[i][j] = 'N';
   }
  }
 }
 return g[l2][l1];
}

int counter;
void PathRecovery(int x, int y, int *delta, char *src, char *dst){
 int ndelta;

 if(x == 0 && y == 0){
  *delta = 0;
  return;
 }
 else {
  switch(move[x][y]){
  case 'R':
   PathRecovery(x-1,y-1,&ndelta,src,dst);
   *delta = ndelta;
   printf("%d␣Replace␣%d,%c\n", counter++, y+ndelta, dst[x-1]);
   break;
  case 'I':
   PathRecovery(x-1,y,&ndelta,src,dst);
   *delta = ndelta+1;
   printf("%d␣Insert␣%d,%c\n", counter++, y+ndelta+1, dst[x-1]);
   break;
  case 'D':
   PathRecovery(x,y-1,&ndelta,src,dst);
   *delta = ndelta-1;
   printf("%d␣Delete␣%d\n", counter++, y+ndelta);
   break;
  case 'N':
   PathRecovery(x-1,y-1,&ndelta,src,dst);
   *delta = ndelta;
   break;
  }
 }
}

int main(){
 int x, first = 1, delta;
 char s1[MAXN], s2[MAXN];
```

```
  while(fgets(s1, MAXN, stdin)){
    if(first) first = 0;
    else printf("\n");
    fgets(s2, MAXN, stdin);
    s1[strlen(s1)-1] = 0;
    s2[strlen(s2)-1] = 0;
    if(first) first = 0;
    else printf("\n");
    x = editDistance(s1, s2, 1,1,1);
    printf("%d\n", x);
    counter = 1;
    PathRecovery(strlen(s2), strlen(s1), &delta, s1, s2);
  }
  return 0;
}
```

## Maximum Submatrix Sum

```
/*
  Description: Given a matrix with n rows and m columns, find
           the rectangular submatrix with the largest sum.

  Complexity: O(n*m^2)

  Notes: This code can easily be converted to work with a
       matrix of doubles. Matricies have the coordinate (0,0) in
       the upper left corner and (n-1,m-1) in the lower right.
       Submatricies are inclusive, i.e. the submatrix
       (t=5, b=6, l=3, r=4) contains the elements:
       (5,3), (5,4), (6,3), (6,4).
       The result is not guaranteed to be unique.
*/

/* fix compile warning */

#include <stdio.h>
#define MAXN 100
#define Atype double

typedef struct {
  int top, bot, left, right;
  Atype sum;
} submat_t;

submat_t max_submatrix(Atype vals[MAXN][MAXN], int n, int m) {
  submat_t best, prev[MAXN][MAXN];
  int row, i, j;
  Atype sum;

  for (i = 0; i < n; i++)
    for (j = 0; j <= i; j++) prev[i][j].sum = -1;
  best.sum = vals[0][0];
  best.left = best.right = best.top = best.bot = 0;

  for (row = 0; row < n; row++) {
    for (i = 0; i < m; i++) {
      for (sum = 0, j = i; j < m; j++) {
        sum += vals[row][j];
        if (prev[i][j].sum <= 0) {
          prev[i][j].sum = 0;
          prev[i][j].top = row;
        }
        prev[i][j].sum += sum;
        if (prev[i][j].sum > best.sum) { /* put tie-breaking here */
          best = prev[i][j];
          best.right = j;
          best.left = i;
          best.bot = row;
        }
      }
    }
  }
  return best;
}

/* If you have matricies that are long and skinny (like 7x40 or 100x20)
   call this function instead of max_submatrix. If the problem can be
   solved faster by transposing the matrix, this function will
   automatically determine that and solve the problem. It's usage
   is identical to max_submatrix. */

submat_t max_submat_t(Atype mat[MAXN][MAXN], int n, int m) {
  Atype transp[MAXN][MAXN];
  int i, j, tmp;
  submat_t res;

  if (m <= n) return max_submatrix(mat, n, m);
  for (i = 0; i < n; i++)
    for (j = 0; j < n; j++) transp[j][i] = mat[i][j];
  res = max_submatrix(transp, m, n);
  tmp = res.top;
  res.top = res.left;
  res.left = tmp;
  tmp = res.bot;
```

```
  res.bot = res.right;
  res.right = tmp;
  return res;
}

int main() {
  Atype mat[100][100];
  int nrows, ncols, i, j;
  submat_t max;
  while (scanf("%d␣%d", &nrows, &ncols)==2) {
    for (i = 0; i < nrows; i++)
      for (j = 0; j < ncols; j++)
        scanf("%lf",&mat[i][j]);
    max = max_submatrix(mat,nrows,ncols);
    printf("(%d,%d)␣(%d,%d)␣has␣a␣sum␣of␣%g.\n",
           max.top, max.left, max.bot, max.right, max.sum);
  }
  return 0;
}
```

## Edit distance between two strings

```
/*
 * The edit distance between two strings is determined based on
 * three possible actions:
 * 1) Change: changes one symbol in a string to match
 * a symbol in another string.
 * 2) Insert: inserts a symbol into a string at a position
 * corresponding to the same symbol in the other string.
 * 3) Delete: deletes a symbol from one string that doesn't
 * appear in the other string.
 *
 * There is a simple (very simple) example for path recovery. Should
 * you wish to implement your own, the parts of this code pertaining to
 * path recovery are centralized in the procedure path_recovery().
 *
 * This algorithm uses dynamic programming and is O(nm). In theory,
 * ie. not in this implementation, it is not necessary to store the
 * entire table, only two columns need to be stored at any given step
 * if all that is desired is to compute the edit distance. In this
 * case, it takes O(m) space. NOTE: If this is done, path recovery
 * is NOT possible.
 *
 * If you don't like this algorithm, another possiblity is to use
 * Dijkstra's algorithm for shortest paths in the grid graph.
 *
 * Two possible edit distance configurations are coded below, and
 * the vast multitude remaining are left as an exercise for the
 * reader.
 *
 * - int edit_general(char *A, char *B, int change, int insert, int delete);
 * Specify the strings and assign weights for changing, inserting,
 * and deleting.
 *
 * - int edit_equal(char *A, char *B);
 * Specify the strings. Changing, inserting, and deleting all
 * carry unit weight.
 *
 * - int edit_no_change(char *A, char *B);
 * Specify the strings. Changing is not allowed. To represent
 * this to the algorithm, the weight of changing is set to greater
 * than the sum of insert and delete--in this case, three.
 */

#include <stdio.h>
#include <stdlib.h>
#include <assert.h>

#define INSERT 1
#define DELETE 2
#define CHANGE 3

typedef struct {
  /* Empty struct to hold a sample to return, if desired. */
  /* Note that you have to code your own Return, so that the
     format will be correct */

  int *mods; /* Stores the modifications made, in reverse order */
  int len; /* Stores the number of modifications. */
} ReturnType;


int is_insert (int **EDIT, int i, int j, int insert) {
  if (j && (EDIT[i][j] == EDIT[i][j-1] + insert)) return 1;
  return 0;
}


int is_delete (int **EDIT, int i, int j, int delete) {
  if (i && (EDIT[i][j] == EDIT[i-1][j] + delete)) return 1;
  return 0;
}
```

```
int is_change (int **EDIT, int i, int j, char *A, char *B, int change) {
 if (i && j && (A[i-1] != B[j-1]) && (EDIT[i][j] == EDIT[i-1][j-1] +change)) {
  return 1;
 }
 return 0;
}


int is_nochange (int **EDIT, int i, int j, char *A, char *B) {
 if (i && j && (A[i-1] == B[j-1]) && (EDIT[i][j] == EDIT[i-1][j-1])) return 1;
 return 0;
}


void path_recovery(int **EDIT, char *A, char *B, int change, int insert,
        int delete, ReturnType *aSample) {
 int i, j, k;

 /*
  * At this point, you have a number for changes required to
  * change from one string to the other. If you want a sample
  * returned, this is where you would specify the format.
  *
  * In order to retrieve the path, start at EDIT[strlen(a)][strlen(b)]
  * and travel back up the graph to the start--EDIT[0][0].
  *
  * To determine which position in the string the change (whatever)
  * is taking place is messy. Have fun. :-)
  */

 i = strlen(A);
 j = strlen(B);

 /* Fix up your storage space, if necessary */
 aSample->mods = (int *) malloc (EDIT[i][j] * sizeof(int));
 assert(aSample->mods);
 aSample->len = EDIT[i][j];
 k = EDIT[i][j] - 1; /* We'll use this to index into the
          * storage space we have. Starts from
          * the end and works back.
          */

 while (i || j || k>-1) {
  if (is_insert(EDIT, i, j, insert)) {
   /* Character B[j-1] was inserted */
   aSample->mods[k] = INSERT; k--;

   j--;
  } else if (is_delete(EDIT, i, j, delete)) {
   /* Character A[i-1] was deleted */
   aSample->mods[k] = DELETE; k--;

   i--;
  } else if (is_change(EDIT, i, j, A, B, change)) {
   /* Character A[i-1] was replaced by B[j-1] */
   aSample->mods[k] = CHANGE; k--;

   i--; j--;
  } else if (is_nochange(EDIT, i, j, A, B)) {
   /* Nothing happened */

   i--; j--;
  }
 }
}


int edit_general(char *A, char *B,
         int change, int insert, int delete, ReturnType *aSample) {
 int len_a, len_b; /* Lengths of strings A & B */
 int** EDIT; /* Array to hold path information */
 int i, j;
 int cha, ins, del;
 int number_of_changes;

 len_a = strlen(A);
 len_b = strlen(B);

 EDIT = (int **) malloc ((len_a+1) * sizeof(int *));
 assert(EDIT);

 for (i=0; i<= len_a; i++) {
  EDIT[i] = (int *) malloc ((len_b+1) * sizeof(int));
  assert(EDIT[i]);
 }

 /* More details that already provided...
  *
  * The goal of this exercise is essentially to transform string A
  * into string B through a minimal set of changes. To that end, all
  * deletions, insertions, and changes are from the point of view of
  * string A.
  *
  * For the following, B is considered be represented by the +x axis,
  * and A by the -y axis... that is, B across, A down.
  */

 /*
  * Set the boundary values -- this means that to go from a string
  * of length i to 0 requires a min. of i deletions. Similar for j,
  * except j deletions.
  */
 for (i=0; i<= len_a; i++) {
  EDIT[i][0] = i;
```

```c
  }

  for (j=0; j<= len_b; j++) {
    EDIT[0][j] = j;
  }

  /* Calculate */
  for (i=1; i<= len_a; i++) {
    for (j=1; j<= len_b; j++) {
      /* Calculate the new value */
      cha = EDIT[i-1][j-1];
      ins = EDIT[i][j-1] + insert;
      del = EDIT[i-1][j] + delete;

      /* Are the previous elements the same already? If not, change. */
      if (A[i-1] != B[j-1]) {
    cha += change;
      }

      /* Decide what is best (ie. min(cha, ins, del)) */
      if ((del <= ins) && (del <= cha)) {
    EDIT[i][j] = del;
      } else if (ins <= cha) {
    EDIT[i][j] = ins;
      } else {
    EDIT[i][j] = cha;
      }
    }
  }
  number_of_changes = EDIT[len_a][len_b];

  path_recovery(EDIT, A, B, change, insert, delete, aSample);

  /* Free up the memory. */
  for (i=0; i<= len_a; i++) {
    free(EDIT[i]);
  }

  free(EDIT);

  return number_of_changes;
}
```

```c
int edit_equal(char *A, char *B, ReturnType *aSample) {
  return edit_general(A, B, 1, 1, 1, aSample);
}


int edit_no_changes(char *A, char *B, ReturnType *aSample) {
  return edit_general(A, B, 3, 1, 1, aSample);
}


int main () {
  char A[80];
  char B[80];
  ReturnType aSample;
  int i;

  while (scanf("%s %s", A, B) != EOF) {
    printf("Disallowing changes, it is: %d\n",
      edit_no_changes(A, B, &aSample));
    free(aSample.mods);

    printf("Allowing changes, the distance is: %d\n",
      edit_equal(A, B, &aSample));

    for (i=0; i<aSample.len; i++) {
      switch(aSample.mods[i]) {
      case INSERT:
    printf("Insert\n");
    break;
      case DELETE:
    printf("Delete\n");
    break;
      case CHANGE:
    printf("Change\n");
    break;
      }
    }
    free(aSample.mods);
  }

  return 0;
}
```

## Transpose String

```
#include <stdio.h>
#include <string.h>
#include <limits.h>
#include <assert.h>

typedef enum { Insert = 1, Delete, Replace, Match, Transpose } move_t;

typedef struct {
 move_t move;
 int x,y;
} action_t;

char a[512], b[512];
int m, n;

int dist[512][512];
action_t action[512][512];

int min(int a, int b) { return a > b ? b : a; }
int min3(int a, int b, int c) { return min(a, min(b,c)); }
int min4(int a, int b, int c, int d) { return min(a, min(b, min(c,d))); }

/* returns the smallest number of moves required to transform
   the substring of _a_ starting at _i_ into the substring of _b_
   starting at index _j_ */

int
string_distance(int i, int j) {
 int insert_cost, delete_cost, replace_cost,
    match_cost, transpose_cost, mincost;

 if( i == m && j == n) {
  action[i][j].move = Match;
  action[i][j].x = i; action[i][j].x = j;
  dist[i][j] = 0;
  return 0;
 } else if( i == m ) {

  /* we are at the end of the source string - we must insert */
  dist[i][j] = string_distance(i,j+1) + 1;
  action[i][j].move = Insert;
  action[i][j].x = i; action[i][j].y = j+1;
  return dist[i][j];

 } else if( j == n ) {

  /* we are at the end of the destination string - we must delete */
  dist[i][j] = string_distance(i+1,j) + 1;
  action[i][j].move = Delete;
  action[i][j].x = i+1; action[i][j].y = j;
  return dist[i][j];

 }

 /* if we already know the cost... */
 if(dist[i][j] != INT_MAX) return dist[i][j];

 insert_cost = string_distance(i,j+1) + 1;
 delete_cost = string_distance(i+1,j) + 1;
 replace_cost= string_distance(i+1,j+1) + 1;
 match_cost = a[i] == b[j] ? string_distance(i+1,j+1) : INT_MAX;
 transpose_cost = (i+1 < m && j+1 < n) && (a[i+1] == b[j] && a[i] == b[j+1])
              ? string_distance(i+2,j+2) + 1: INT_MAX;

 mincost = min( insert_cost,
           min(delete_cost,
              min(replace_cost,
                 min(match_cost, transpose_cost))));

 if( match_cost == mincost ) {

  action[i][j].move = Match;
  action[i][j].x = i+1; action[i][j].y = j+1;

 } else if( replace_cost == mincost ) {

  action[i][j].move = Replace;
  action[i][j].x = i+1; action[i][j].y = j+1;

 } else if( delete_cost == mincost ) {

  action[i][j].move = Delete;
  action[i][j].x = i+1; action[i][j].y = j;

 } else if( transpose_cost == mincost ) {

  action[i][j].move = Transpose;
  action[i][j].x = i+2; action[i][j].y = j+2;
 } else {

  action[i][j].move = Insert;
  action[i][j].x = i; action[i][j].y = j+1;

 }

 dist[i][j] = mincost;

 return dist[i][j];
```

```
}


int
main () {
 int i,j,offset,p,q;

 while( fgets(a,512,stdin) != NULL ) {
  fgets(b, 512, stdin);

  /* note that we are saying the string is one character longer than
     it really is. This makes it all work out nicely. The null
     is special character that everything past it matches for free*/

  m = strlen(a);
  n = strlen(b);
  a[m-1] = '\0'; m--;
  b[n-1] = '\0'; n--;

  for( i = 0; i < m; i++ ) {
   for( j = 0; j < n; j++ ) {
    dist[i][j] = INT_MAX;
   }
  }

  printf("distance␣is␣%d.\n", string_distance(0,0));

  i = 0; j = 0;
  offset = 0;
  do {
   switch(action[i][j].move) {
   case Insert:
  printf("Insert␣%c␣before␣position␣%d\n", b[j], i+offset);
  offset++;
  break;
   case Delete:
  printf("Delete␣%c␣at␣position␣%d\n", a[i], i+offset);
  offset--;
  break;
   case Replace:
  printf("Replace␣%c␣at␣position␣%d␣with␣%c\n", a[i], i+offset, b[j]);
  break;
   case Match:
  printf("Match␣%c␣at␣position␣%d\n", a[i], i+ offset);
  break;
   case Transpose:
  printf("Tranpose␣%c%c␣at␣position␣%d\n", a[i], a[i+1], i+offset);
  break;
   default:
  assert(!fprintf(stderr, "whoa␣there.\n"));
   }
   p = i; q = j; /* watch out for simultaneous assignment! duh! */
   i = action[p][q].x; j = action[p][q].y;
  } while( i != m || j != n);


 }
 return 0;
}
```

## Miscellaneous

### Subsets of a subset in $O(3^n)$

```
for (int i=0; i < (1<<n); ++i){
   for (int i2 = i; i2 > 0; i2 = (i2-1) & i){
   }
}
```

### Fractional Cascading

Given $k$ sorted sequences $S$, find first element $\geq q$ in all $k$ sequences in $O(logn + k)$.
We will generate $k$ modified sequences $M$. Every element of a sequence $M_i$ will have two indexes associated with it: it's supposed index in $S_i$ rounding to the largest option, and it's supposed index in $M_{i+1}$.
Starting in $M_k(M_k = S_k$, all the second indexes are zero, and the first indexes are the actual indexes in $S_i$), for every $M_i$ do:

Insert in $M_{i-1}$ all the elements in odd positions of $M_i$

In every insertion maintain the index in $M_i$ of the last inserted element, and the index of the element currently being inserted. With that information, find out the second indexes of the elements between the last inserted and the currently being inserted.

The first indexes of all the original elements are initialized with their actual indexes, and the first indexes of the inserted elements are initialized with min(index of first original element before the inserted one +1, sequence size - 1) or if the element is being inserted in position 0, the index is 0

With that information, do a binary search in the first sequence ($M_0$), with that you'll find the answer to the query in $S_0$ being $S_{0element.first}$, and then you go to the position element.second of $M_1$, and check if element in position element.second-1 is $\geq q$, if it is true, the element you're looking for is element.second-1, otherwise it's element.second. The answer to the current sequence will be in the index that is in the element in the found index (element.second-1 or element.second) .first. Repeat the process until you find the last answer in $M_k$

### Warnsdorff's heuristic for knight's tour

At each step choose a square which has the least number of valid moves that the knight can make from there.

### Optimal BST - Knuth Optimization

$root[i, j-1] \leq root[i, j] \leq root[i+1, j]$.

### Flow-shop scheduling (Johnson's problem)

Schedule $N$ jobs on 2 machines to minimize completition time. $i$-th job takes $a_i$ and $b_i$ time to execute on 1st and 2nd machine, respectively. Each job must be first executed on the first machine, then on second. Both machines execute all jobs in the same order. Solution: sort jobs by key $a_i < b_i ? a_i : (\infty - b_i)$, i.e. first execute all jobs with $a_i < b_i$ in order of increasing $a_i$, then all other jobs in order of decreasing $b_i$.

## Generate de Bruijin sequence

```
string seq;

int pw(int b,int a){
   int ans = 1;
   while ( a ){
      if (a&1) ans *= b;
      b *= b;
      a /= 2;
   }
   return ans;
}

void debruijn( int n, int k ){
   seq = "";
   char s[n];
   if ( n == 1 ){
      for ( int i = 0; i < k; i++ )
         seq += char('0'+i);
   }
   else{
      for ( int i = 0; i < n-1; i++ )
         s[i] = k-1;

      int kn = pw(k, n-1);
      char nxt[kn]; memset(nxt, 0, sizeof(nxt));
      kn *= k;
      for ( int h = 0; h < kn; h++ ){
         int m = 0;
         for ( int i = 0; i < n-1; i++ ){
            m *= k;
            m += s[(h+i)%(n-1)];
         }
         seq += char('0'+nxt[m]);
         s[h%(n-1)] = nxt[m];
         nxt[m]++;
      }
   }
}
```

A k-ary De Bruijn sequence B(k, n) of order n is a cyclic sequence of a given alphabet A with size k for which every possible subsequence of length n in A appears as a sequence of consecutive characters exactly once.

## Josephus

```
int live[MAXN];
void josephus( int n, int m ){ // n people, m-th get killed
   live[1] = 0;
   for( int i = 2; i <= n; i++ )
      live[i] = (live[i-1]+(m%i))%i;
}
```

## LIS O(n log n)

```
vector<int> lis(vector<int>& seq){
   int smallest_end[seq.size()+1], prev[seq.size()];
   smallest_end[1] = 0;

   int sz = 1;
   for (int i = 1; i < seq.size(); ++i){
      int lo = 0, hi = sz;
      while (lo < hi){
         int mid = (lo + hi + 1)/2;
         if (seq[smallest_end[mid]] <= seq[i])
            lo = mid;
         else
            hi = mid - 1;
      }
```

```
        prev[i] = smallest_end[lo];
        if (lo == sz)
            smallest_end[++sz] = i;
        else if (seq[i] < seq[smallest_end[lo+1]])
            smallest_end[lo+1] = i;
    }
    vector<int> ret;
    for (int cur = smallest_end[sz]; sz > 0; cur = prev[cur], --sz)
        ret.push_back(seq[cur]);
    reverse(ret.begin(), ret.end());

    return ret;
}
```

## Fast Input

```
int next_int(){
    int n = 0, neg = 1;
    char c = getchar_unlocked();
    if ( c == EOF ) exit(0);
    while (!('0' <= c && c <= '9')) {
        if ( c == '-' ) neg = -1;
        c = getchar_unlocked();
        if ( c == EOF ) exit(0);
    }
    while ('0' <= c && c <= '9') {
        n = n * 10 + c - '0';
        c = getchar_unlocked();
    }
    return neg*n;
}
```

## Fraction

```
struct frac {
    long long num, den;
    frac(long long num = 0, long long den = 1) { set_val(num, den); }

    void set_val(long long _num, long long _den) {
        num = _num/__gcd(_num, _den);
        den = _den/__gcd(_num, _den);
        if(den < 0) { num *= -1; den *= -1; }
    }

    void operator*=(frac f) { set_val(num * f.num, den * f.den); }
    void operator+=(frac f) { set_val(num * f.den + f.num * den, den * f.den); }
    void operator-=(frac f) { set_val(num * f.den - f.num * den, den * f.den); }
    void operator/=(frac f) { set_val(num * f.den, den * f.num); }
};
bool operator==(frac a, frac b) { return a.num * b.den == b.num * a.den; }
bool operator!=(frac a, frac b) { return !(a == b); }
bool operator<(frac a, frac b) { return a.num * b.den < b.num * a.den; }
bool operator<=(frac a, frac b) { return (a == b) || (a < b); }
bool operator>(frac a, frac b) { return !(a <= b); }
bool operator>=(frac a, frac b) { return !(a < b); }
frac operator/(frac a, frac b) { frac ret = a; ret /= b; return ret; }
frac operator*(frac a, frac b) { frac ret = a; ret *= b; return ret; }
frac operator+(frac a, frac b) { frac ret = a; ret += b; return ret; }
frac operator-(frac a, frac b) { frac ret = a; ret -= b; return ret; }
frac operator-(frac f) { return 0 - f; }

std::ostream& operator<<(std::ostream& o, const frac f) {
    o << f.num << "/" << f.den;
    return o;
}
```