ACM ICPC Team Reference
2010 World Finals

Team Anuncie Aqui
Universidade Federal de Sergipe

1. Configuration files and scripts

1.1. .emacs. Hash: c4c6b75b731e46e642e98db153594c25

```
(global-font-lock-mode t)
(setq transient-mark-mode t)
(require 'font-lock)
(require␣'paren)
(global-set-key [f5] 'cxx-compile)
(set-input-mode␣nil␣nil␣1)
(fset␣'yes-or-no-p 'y-or-n-p)

(require␣'cc-mode)
(defun cxx-compile()
  (interactive)
  (progn
   (save-buffer)
   (compile (concat "g++␣-g␣-O2␣-o␣" (substring buffer-file-name 0 -4)
         buffer-file-name))
  )
)

(add-hook 'c++-mode-hook␣'(lambda () (c-set-style "stroustrup")))
```

1.2. .vimrc. Hash: c1e8578e5f779285977a53cce7a48031

```
syn on
filetype on
filetype plugin on
filetype indent on
colorscheme koehler

set number
set shiftwidth=4
set ts=4

imap <C-Space> <C-P>
set cinkeys=0{,0},0),0#,!<Tab>,;,:,o,O,e
set indentkeys=!<Tab>,o,O

runtime mswin.vim
```

1.3. **Hash generator.** Hash: 0d22aecd779fc370b30a2c628aff517c

```
#!/bin/sh
sed ':a;N;$!ba;s/[ \n\t]//g' | md5sum | cut -d' ' -f1
```

1.4. **Solution template.** Hash: 91b0fffaa0504c01fe4cc05bc08561d0

```
#include <iostream>
#include <cstdio>
#include <algorithm>
#include <vector>
#include <set>
#include <queue>
#include <map>
#include <utility>
#include <cstring>
#include <cstdlib>
#include <cmath>
#include <cassert>
using namespace std;

typedef double TYPE;
const TYPE EPS = 1e-9;
const TYPE 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); }

int main() {
   return 0;
}
```

2. Graph algorithms

2.1. **Tarjan's SCC algorithm.** Hash: f98d9589db68c8f1e8274cf53eb7f3bf

```
int lowest[MAXV], num[MAXV], visited[MAXV], comp[MAXV];
int prev_edge[MAXE], last_edge[MAXV], adj[MAXE], nedges;
int cur_num, cur_comp;
stack<int> visiting;

int t_init() {
   memset(last_edge, -1, sizeof last_edge);
   nedges = 0;
}

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

int tarjan_dfs(int v) {
   lowest[v] = num[v] = cur_num++;
   visiting.push(v);
   visited[v] = 1;
   for(int i = last_edge[v]; i != -1; i = prev_edge[i]) {
      int w = adj[i];
      if(visited[w] == 0) lowest[v] = min(lowest[v], tarjan_dfs(w));
      else if(visited[w] == 1) lowest[v] = min(lowest[v], num[w]);
   }

   if(lowest[v] == num[v]) {
      int last = -1;
      while(last != v) {
         comp[last = visiting.top()] = cur_comp;
         visited[last] = 2;
         visiting.pop();
      }
      cur_comp++;
   }

   return lowest[v];
```

```
}

void tarjan_scc(int num_v = MAXV) {
   visiting = stack<int>();
   memset(visited, 0, sizeof visited);
   cur_num = cur_comp = 0;

   for(int i = 0; i < num_v; i++)
      if(!visited[i])
         tarjan_dfs(i);
}
```

2.2. **Dinic's algorithm.** Hash: 4dd537effe7e233681c099912397839a

```
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;
}

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

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[rev(i)] -= 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;
}
```

2.3. **Busacker-Gowen's algorithm.** Hash: 6933692fe046f78da13b05166c7e6d23

```
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;

inline 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, -cost, true);
}

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

inline 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 retFlow = 0, retCost = 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];
         retCost += cost[rev(bg_prev[cur])] * capres[sink];
         cur = adj[bg_prev[cur]];
      }
      retFlow += 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(retFlow, retCost);
}
```

2.4. **Gabow's algorithm.** Hash: 31f8b67cd2b16187c6733f42801ee2be

```
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) {
   prev_edge[nedges] = last_edge[a];
   v[nedges] = a;
   w[nedges] = b;
   last_edge[a] = nedges++;

   prev_edge[nedges] = last_edge[b];
   v[nedges] = b;
   w[nedges] = a;
   last_edge[b] = nedges++;
}

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 u = 0, 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(true) {
               if(s != -1) swap(r, s);
               r = first[label[mate[r]]];
               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;
   }
```

3. Math

3.1. **Fractions.** Hash: 379fd408c3007c650c022fd4adfeabbd

```
struct frac {
   long long num, den;

   frac() : num(0), den(1) { };
   frac(long long num, long long den) { set_val(num, den); }
   frac(long long num) : num(num), den(1) { };

   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) {
   if((a.den < 0) ^ (b.den < 0)) return a.num * b.den > b.num * a.den;
   return a.num * b.den < b.num * a.den;
}

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

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 == 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; }
```

3.2. **Chinese remainder theorem.** Hash: 06b5ebd5c44c204a4b11bbb76d09023d

```
struct t {
   long long a, b; int g;
   t(long long a, long long b, int g) : a(a), b(b), g(g) { }
   t swap() { return t(b, a, g); }
};

t egcd(int p, int q) {
   if(q == 0) return t(1, 0, p);
```

```
   t t2 = egcd(q, p % q);
   t2.a -= t2.b * (p/q);
   return t2.swap();
}
```

```
int crt(int a, int p, int b, int q) {
   t t2 = egcd(p, q); t2.a %= p*q; t2.b %= p*q;
   assert(t2.g == 1);
   int ret = ((b * t2.a)%(p*q) * p + (a * t2.b)%(p*q) * q) % (p*q);
   return ret >= 0 ? ret : ret + p*q;
}
```

3.3. **Longest increasing subsequence.** Hash: 0f80b5d3af188d8bf4d1cbe45a76b46d

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

3.4. **Simplex (Warsaw University).** Hash: c687094970cf1953fd6f87a01adc6a95

```
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;
inline 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;
}
```

3.5. **Romberg's method.** Hash: a85facba1eac60c8909b04b552bd2222

```
long double romberg(int a, int b, double(*func)(double)) {
   long double approx[2][50];
   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];
}
```

3.6. **Floyd's cycle detection algorithm.** Hash: 4aaa3277ea9011cae6d9b1358521f02c

```
pair<int, int> floyd(int x0) {
   int t = f(x0), h = f(f(x0)), start = 0, length = 1;
   while(t != h) t = f(t), h = f(f(h));
   h = t; t = x0;
   while(t != h) t = f(t), h = f(h), start++;
   h = f(t);
   while(t != h) h = f(h), length++;
   return make_pair(start, length);
}
```

3.7. **Pollard's rho algorithm.** Hash: ad4ee1d4afc564b2c55f90d6269994c4

```
long long pollard_r, pollard_n;

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

long long pollard(long long n) {
   srand(unsigned(time(0)));
   pollard_n = n;
   long long d = 1;
   do {
      d = 1;
      pollard_r = rand() % n;

      long long 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;
}
```

3.8. **Miller-Rabin's algorithm.** Hash: 3a5ca9acc192107c3eb9940088ad16c7

```
int fastpow(int base, int d, int n) {
   int ret = 1;
   for(long long 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 = (long long)base_2r * base_2r % n;
   }

   return false;
}

int isprime(int n) {
   return miller_rabin(n, 2) && miller_rabin(n, 7) && miller_rabin(n, 61);
}
```

3.9. **Polynomials (PUC-Rio).** Hash: d69d1ad494e487327d2338e69eccfa2f

```
typedef complex<double> cdouble;
int cmp(cdouble x, cdouble y = 0) {
  return cmp(abs(x), abs(y));
}
const int TAM = 200;
struct poly {
  cdouble poly[TAM]; int n;
  poly(int n = 0): n(n) { memset(p, 0, sizeof(p)); }
  cdouble& operator [](int i) { return p[i]; }
  poly operator ~() {
    poly r(n-1);
    for (int i = 1; i <= n; i++)
      r[i-1] = p[i] * cdouble(i);
    return r;
  }
  pair<poly, cdouble> ruffini(cdouble z) {
    if (n == 0) return make_pair(poly(), 0);
    poly r(n-1);
    for (int i = n; i > 0; i--) r[i-1] = r[i] * z + p[i];
    return make_pair(r, r[0] * z + p[0]);
  }
  cdouble operator ()(cdouble z) { return ruffini(z).second; }
  cdouble find_one_root(cdouble x) {
    poly p0 = *this, p1 = ~p0, p2 = ~p1;
    int m = 1000;
    while (m--) {
      cdouble y0 = p0(x);
      if (cmp(y0) == 0) break;
      cdouble G = p1(x) / y0;
      cdouble H = G * G - p2(x) - y0;
      cdouble R = sqrt(cdouble(n-1) * (H * cdouble(n) - G * G));
      cdouble D1 = G + R, D2 = G - R;
      cdouble a = cdouble(n) / (cmp(D1, D2) > 0 ? D1 : D2);
      x -= a;
      if (cmp(a) == 0) break;
    }
    return x;
  }
```

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

4. Geometry

4.1. **Point class.** Hash: 66e85d5b140956c47aa31754eab18864

```
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 << ")";
}
```

4.2. **Intersection primitives.** Hash: ab780978106a5c062b8f7a129ebc9196

```
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);
}
```

4.3. **Polygon primitives.** Hash: fba20bb1645bf37ab6c9b309d3850a7d

```
double p_area(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;
}
```

4.4. **Miscellaneous primitives.** Hash: be051245293a9db9c991d414c598e854

```
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; }
```

4.5. **Smallest enclosing circle.** Hash: 4e41d94c106dee349b45ca542ff0a532

```
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;
}
```

4.6. **Convex hull.** Hash: a7f921d07f1b9b8a0053a0833329ddcf

```
pt pivot;

bool hull_comp(pt a, pt b) {
   int turn = ccw(a, b, pivot);
```

```
   return turn == 1 || (turn == 0 && cmp(norm(a), norm(b)) < 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;
}
```

4.7. **Closest pair of points.** Hash: 251ad75a3af2d531a0cbb4e8138d3aef

```
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 <= 15 && 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);
}
```

4.8. **Kd-tree.** Hash: 4a0b6323567d651c20409c782292db8e

```
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) {
    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;

    int ret = 0;
    if(x) {
        if(a <= split[root])
            ret += kd_query(2*root + 1, a, b, c, d,
                    ca, min(cb, split[root]), cc, cd, !x);
        if(split[root] <= b)
            ret += kd_query(2*root + 2, a, b, c, d,
                    max(ca, split[root]), cb, cc, cd, !x);
    }
    else {
        if(c <= split[root])
            ret += kd_query(2*root + 1, a, b, c, d,
                    ca, cb, cc, min(cd, split[root]), !x);
        if(split[root] <= d)
            ret += kd_query(2*root + 2, a, b, c, d,
                    ca, cb, max(cc, 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(2e9, 2e9) : 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;
}
```

4.9. **Range tree.** Hash: 0a873f00972df011cf51987924ac0ee6

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

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

      sort(tree[root].begin(), tree[root].end(), compy);
      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);

   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 < tree[cl].size() && compy(tree[cl][llink], last))
         llink++;
      while(rlink < tree[cr].size() && 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());
   for(int i = 0; i < pts.size(); i++) xs.push_back(pts[i].x);
   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(-INF, c), compy)
         - tree[0].begin();
      posr = upper_bound(tree[0].begin(), tree[0].end(), pt(INF, d), compy)
         - tree[0].begin();
   }

   if(a <= xs[l] && xs[r] <= b)
      return posr - posl;
   if(posl >= tree[root].size()) return 0;

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

5. Data structures

5.1. **Treap.** Hash: 2199b72803301716616a462d9d5e9a66

```
typedef int TYPE;

class treap {
public:
   treap *left, *right;
   int priority, sons;
```

```
   TYPE value;

   treap(TYPE value) : left(NULL), right(NULL), value(value), sons(0) {
      priority = rand();
   }

   ~treap() {
      if(left) delete left;
      if(right) delete right;
   }
};

treap* find(treap* t, TYPE val) {
   if(!t) return NULL;
   if(val == t->value) return t;

   if(val < t->value) return find(t->left, val);
   if(val > t->value) return find(t->right, val);
}

void rotate_to_right(treap* &t) {
   treap* n = t->left;
   t->left = n->right;
   n->right = t;
   t = n;
}

void rotate_to_left(treap* &t) {
   treap* n = t->right;
   t->right = n->left;
   n->left = t;
   t = n;
}

void fix_augment(treap* t) {
   if(!t) return;
   t->sons = (t->left ? t->left->sons + 1 : 0) +
      (t->right ? t->right->sons + 1 : 0);
}

void insert(treap* &t, TYPE val) {
   if(!t)
      t = new treap(val);
   else
      insert(val <= t->value ? t->left : t->right, val);

   if(t->left && t->left->priority > t->priority)
      rotate_to_right(t);
   else if(t->right && t->right->priority > t->priority)
      rotate_to_left(t);

   fix_augment(t->left); fix_augment(t->right); fix_augment(t);
}

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

void erase(treap* &t, TYPE val) {
   if(!t) return;

   if(t->value != val)
      erase(val < t->value ? t->left : t->right, val);
   else {
      if(!t->left && !t->right)
         delete t, t = NULL;
      else {
         p(t->left) < p(t->right) ? rotate_to_left(t) : rotate_to_right(t);
         erase(t, val);
      }
   }

   fix_augment(t->left); fix_augment(t->right); fix_augment(t);
}
```

5.2. **Heap.** Hash: e334218955a73d1286ad0fc19e84b642

```
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;
        }
    }
};
```

5.3. **Big numbers (PUC-Rio).** Hash: a7d74e7158634f9201c19235badd3364

```
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;
}
};
```

6. String algorithms

6.1. **Manber-Myers' algorithm.** Hash: b32cb670595bef320decbceed7420bb8

```
int pos[MAXSZ], prm[MAXSZ], cnt[MAXSZ];
bool bh[MAXSZ + 1], b2h[MAXSZ];
int blast[256], bprev[MAXSZ];
int mm_segtree[4*MAXSZ];
string mm_s;

inline void regen_pos(int sz) {
   for(int i = 0; i < sz; i++)
      pos[prm[i]] = i;
}

inline void bubbleupbucket(int index) {
   if(index < 0) return;

   int& prm_ext = prm[index];
   cnt[prm_ext]++;
   prm_ext += cnt[prm_ext] - 1;
   b2h[prm_ext] = true;
}

void updatetree(int root, int l, int r, int pos, int val) {
   if(l == r) { mm_segtree[root] = val; return; }

   int m = (l + r + 1)/2;
   if(pos < m) updatetree(2*root + 1, l, m - 1, pos, val);
   else updatetree(2*root + 2, m, r, pos, val);

   mm_segtree[root] = min(mm_segtree[2*root + 1], mm_segtree[2*root + 2]);
}

int querytree(int root, int l, int r, int begin, int end) {
   if(begin == l && end == r) return mm_segtree[root];

   int m = (l + r + 1)/2;
   if(begin < m && end < m)
      return querytree(2*root + 1, l, m - 1, begin, end);
   else if(begin >= m && end >= m)
      return querytree(2*root + 2, m, r, begin, end);
   else return min(querytree(2*root + 1, l, m - 1, begin, m - 1),
               querytree(2*root + 2, m, r, m, end));
}

void mm_build(string s) {
   mm_s = s;
   memset(blast, -1, sizeof blast);
   memset(bh, 0, sizeof(bool) * s.size());
   memset(mm_segtree, 0x3f, sizeof(int) * 4 * s.size());
   updatetree(0, 0, s.size() - 1, s.size() - 1, 0);

   for(int i = 0; i < s.size(); i++) {
      bprev[i] = blast[s[i]];
      blast[s[i]] = i;
   }
   int let_count = 0;
   for(int i = 0; i < 256; i++) {
      if(blast[i] != -1) {
         bh[let_count] = true;
         if(let_count > 0)
            updatetree(0, 0, s.size() - 1, let_count - 1, 0);
      }
      for(int j = blast[i]; j != -1; j = bprev[j])
         prm[j] = let_count++;
   }
   regen_pos(s.size());
   bh[s.size()] = true;

   for(int st = 1; st < s.size(); st *= 2) {
      memset(cnt, 0, sizeof(int) * s.size());
      memset(b2h, 0, sizeof(bool) * s.size());

      for(int bl = 0, br = 0; br < s.size(); bl = br++)
         for(; !bh[br]; br++)
            prm[pos[br]] = bl;

      bubbleupbucket(s.size() - st);
      for(int bl = 0, br = 0; br < s.size(); bl = br) {
         bubbleupbucket(pos[bl] - st);
         for(br++; !bh[br]; br++)
            bubbleupbucket(pos[br] - st);

         for(int i = bl; i < br; i++) {
            if(pos[i] - st < 0) continue;
            int prm_ext = prm[pos[i] - st];
            if(b2h[prm_ext])
               for(int j = prm_ext + 1; !bh[j] && b2h[j]; j++)
                  b2h[j] = false;
```

```
        }
      }

      regen_pos(s.size());
      for(int i = 0; i < s.size(); i++)
        if(!bh[i] && b2h[i]) {
          bh[i] = true;
          if(pos[i - 1] + st < s.size() && pos[i] + st < s.size()) {
            int m = min(prm[pos[i - 1] + st], prm[pos[i] + st]);
            int M = max(prm[pos[i - 1] + st], prm[pos[i] + st]);
            updatetree(0, 0, s.size() - 1, i - 1,
                       st + querytree(0, 0, s.size() - 1, m, M - 1));
          }
          else
            updatetree(0, 0, s.size() - 1, i - 1, st);
        }
    }
}

inline int lcp(string& s1, int p1, string& s2, int p2) {
  int limit = min(s1.size() - p1, s2.size() - p2), i;
  for(i = 0; i < limit; i++) if(s1[p1 + i] != s2[p2 + i]) break;
  return i;
}

pair<bool, int> mm_find(string s) {
  int l = lcp(mm_s, pos[0], s, 0);
  int r = lcp(mm_s, pos[mm_s.size() - 1], s, 0);

  if(l == s.size() || s[l] < mm_s[pos[0] + l])
    return make_pair(l == s.size(), pos[0]);
  else if(r == s.size() || s[r] > mm_s[pos[mm_s.size() - 1] + r])
    return make_pair(r == s.size(), pos[mm_s.size() - 1]);

  int low = 0, high = mm_s.size() - 1, next, st_n = 0, c_lcp;
  while(high - low > 1) {
    int mid = (low + high)/2;
    c_lcp = max(l, r);
    st_n = 2*st_n + 1 + (l < r);

    if(mm_segtree[st_n] >= c_lcp)
      next = c_lcp + lcp(mm_s, pos[mid] + c_lcp, s, c_lcp);
    else
      next = mm_segtree[st_n];

    if(next == s.size())
      return make_pair(true, pos[mid]);
    else if(s[next] > mm_s[pos[mid] + next]) {
      low = mid;
      l = next;
    }
    else {
      high = mid;
      r = next;
    }
  }

  return make_pair(false, pos[high]);
}
```

6.2. **Morris-Pratt's algorithm.** Hash: ace505eff2be640ff01d7c48b2b7d12f

```
int pi[MAXSZ], res[MAXSZ], nres;

void morris_pratt(string text, string pattern) {
 nres = 0;
 pi[0] = -1;
 for(int i = 1; i < pattern.size(); i++) {
  pi[i] = pi[i-1];
  while(pi[i] >= 0 && pattern[pi[i] + 1] != pattern[i])
    pi[i] = pi[pi[i]];
  if(pattern[pi[i] + 1] == pattern[i]) pi[i]++;
 }
 int k = 0; //k + 1 eh o tamanho do match atual
 for(int i = 0; i < text.size(); i++) {
  while(k >= 0 && pattern[k + 1] != text[i])
    k = pi[k];
  if(pattern[k + 1] == text[i]) k++;
  if(k + 1 == pattern.size()) {
   res[nres++] = i;
   k = pi[k];
  }
 }
}
```

7. Useful mathematical facts

7.1. Prime counting function ($\pi(x)$). The prime counting function is asymptotic to $\frac{x}{\log x}$, by the prime number theorem.

x	10	10^2	10^3	10^4	10^5	10^6	10^7	10^8
$\pi(x)$	4	25	168	1.229	9.592	78.498	664.579	50.847.534

7.2. Partition function. The partition function $p(x)$ counts show many ways there are to write the integer x as a sum of integers.

x	36	37	38	39	40	41	42
p(x)	17.977	21.637	26.015	31.185	37.338	44.583	53.174
x	43	44	45	46	47	100	
p(x)	63.261	75.175	89.134	105.558	125.754	190.569.292	

7.3. Catalan numbers. Catalan numbers are defined by the recurrence:

$$C_{n+1} = \sum_{i=0}^{n} C_i C_{n-i}$$

A closed formula for Catalan numbers is:

$$C_n = \frac{1}{n+1}\binom{2n}{n} = \binom{2n}{n} - \binom{2n}{n+1}$$

7.4. Stirling numbers of the first kind. These are the number of permutations of I_n with exactly k disjoint cycles. They obey the recurrence:

$$\begin{bmatrix} n \\ k \end{bmatrix} = (n-1)\begin{bmatrix} n-1 \\ k \end{bmatrix} + \begin{bmatrix} n-1 \\ k-1 \end{bmatrix}$$

7.5. Stirling numbers of the second kind. These are the number of ways to partition I_n into exactly k sets. They obey the recurrence:

$$\begin{Bmatrix} n \\ k \end{Bmatrix} = k\begin{Bmatrix} n-1 \\ k \end{Bmatrix} + \begin{Bmatrix} n-1 \\ k-1 \end{Bmatrix}$$

A "closed" formula for it is:

$$\begin{Bmatrix} n \\ k \end{Bmatrix} = \frac{1}{k!}\sum_{j=0}^{k}(-1)^{k-j}\binom{k}{j}j^n$$

7.6. Bell numbers. These count the number of ways to partition I_n into subsets. They obey the recurrence:

$$\mathcal{B}_{n+1} = \sum_{k=0}^{n}\binom{n}{k}\mathcal{B}_k$$

x	5	6	7	8	9	10	11	12
$\mathcal{B}_x$	52	203	877	4.140	21.147	115.975	678.570	4.213.597

7.7. Turán's theorem. No graph with n vertices that is K_{r+1}-free can have more edges than the Turán graph: A k-partite complete graph with sets of size as equal as possible.

7.8. Generating functions. A list of generating functions for useful sequences:

$(1, 1, 1, 1, 1, 1, \ldots)$	$\frac{1}{1-z}$
$(1, -1, 1, -1, 1, -1, \ldots)$	$\frac{1}{1+z}$
$(1, 0, 1, 0, 1, 0, \ldots)$	$\frac{1}{1-z^2}$
$(1, 0, \ldots, 0, 1, 0, 1, 0, \ldots, 0, 1, 0, \ldots)$	$\frac{1}{1-z^2}$
$(1, 2, 3, 4, 5, 6, \ldots)$	$\frac{1}{(1-z)^2}$
$(1, \binom{m+1}{m}, \binom{m+2}{m}, \binom{m+3}{m}, \ldots)$	$\frac{1}{(1-z)^{m+1}}$
$(1, c, \binom{c+1}{2}, \binom{c+2}{3}, \ldots)$	$\frac{1}{(1-z)^c}$
$(1, c, c^2, c^3, \ldots)$	$\frac{1}{1-cz}$
$(0, 1, \frac{1}{2}, \frac{1}{3}, \frac{1}{4}, \ldots)$	$\ln\frac{1}{1-z}$

A neat manipulation trick is:

$$\frac{1}{1-z}G(z) = \sum_n \sum_{k \le n} g_k z^n$$

7.9. **The twelvefold way (from Stanley).** How many functions $f\colon N \to X$ are there?

N	X	Any f	Injective	Surjective
dist.	dist.	x^n	$(x)_n$	$x!\left\{{n \atop x}\right\}$
indist.	dist.	$\binom{x+n-1}{n}$	$\binom{x}{n}$	$\binom{n-1}{n-x}$
dist.	indist.	$\left\{{n \atop 1}\right\} + \ldots + \left\{{n \atop x}\right\}$	$[n \leq x]$	$\left\{{n \atop k}\right\}$
indist.	indist.	$p_1(n) + \ldots p_x(n)$	$[n \leq x]$	$p_x(n)$

Where $\binom{a}{b} = \frac{1}{b!}(a)_b$ and $p_x(n)$ is the number of ways to partition the integer n using x summands.

7.10. **Table of non-trigonometric integrals.** Some useful integrals are:

$\int \frac{dx}{x^2+a^2}$	$\frac{1}{a}\arctan\frac{x}{a}$
$\int \frac{dx}{x^2-a^2}$	$\frac{1}{2a}\ln\frac{x-a}{x+a}$
$\int \frac{dx}{a^2-x^2}$	$\frac{1}{2a}\ln\frac{a+x}{a-x}$
$\int \frac{dx}{\sqrt{a^2-x^2}}$	$\arcsin\frac{x}{a}$
$\int \frac{dx}{\sqrt{x^2-a^2}}$	$\ln\left(u+\sqrt{x^2-a^2}\right)$
$\int \frac{dx}{x\sqrt{x^2-a^2}}$	$\frac{1}{a}\operatorname{arcsec}\left\lvert\frac{u}{a}\right\rvert$
$\int \frac{dx}{x\sqrt{x^2+a^2}}$	$-\frac{1}{a}\ln\left(\frac{a+\sqrt{x^2+a^2}}{x}\right)$
$\int \frac{dx}{x\sqrt{a^2+x^2}}$	$-\frac{1}{a}\ln\left(\frac{a+\sqrt{a^2-x^2}}{x}\right)$

7.11. **Table of trigonometric integrals.** A list of common and not-so-common trigonometric integrals:

$\int \tan x dx$	$-\ln\lvert\cos x\rvert$
$\int \cot x dx$	$\ln\lvert\sin x\rvert$
$\int \sec x dx$	$\ln\lvert\sec x + \tan x\rvert$
$\int \csc x dx$	$\ln\lvert\csc x - \cot x\rvert$
$\int \sec^2 x dx$	$\tan x$
$\int \csc^2 x dx$	$\cot x$
$\int \sin^n x dx$	$\frac{-\sin^{n-1} x\cos x}{n} + \frac{n-1}{n}\int \sin^{n-2} x dx$
$\int \cos^n x dx$	$\frac{\cos^{n-1} x\sin x}{n} + \frac{n-1}{n}\int \cos^{n-2} x dx$
$\int \arcsin x dx$	$x\arcsin x + \sqrt{1-x^2}$
$\int \arccos x dx$	$x\arccos x - \sqrt{1-x^2}$
$\int \arctan x dx$	$x\arctan x - \frac{1}{2}\ln\lvert 1-x^2\rvert$

7.12. **Common substitutions.** And finally, a list of common substitutions:

$\int F(\sqrt{ax+b})dx$	$u = \sqrt{ax+b}$	$\frac{2}{a}\int uF(u)du$
$\int F(\sqrt{a^2-x^2})dx$	$x = a\sin u$	$a\int F(a\cos u)\cos u du$
$\int F(\sqrt{x^2+a^2})dx$	$x = a\tan u$	$a\int F(a\sec u)\sec^2 u du$
$\int F(\sqrt{x^2-a^2})dx$	$x = a\sec u$	$a\int F(a\tan u)\sec u\tan u du$
$\int F(e^{ax})dx$	$u = e^{ax}$	$\frac{1}{a}\int \frac{F(u)}{u}du$
$\int F(\ln x)dx$	$u = \ln x$	$\int F(u)e^u du$