# ACM ICPC Team Reference
# 2010 World Finals

Team Anuncie Aqui
Universidade Federal de Sergipe

## 1. Configuration files and scripts

### 1.1. .emacs. Hash: b1040cede72bb06f9b3197eba2d833f5

```
(global-font-lock-mode t)
(setq transient-mark-mode t)

(global-set-key [f5] 'cxx-compile)
(defun cxx-compile()
  (interactive)
  (save-buffer)
  (compile (concat "g++␣-g␣-O2␣-o␣" (file-name-sans-extension buffer-file-name)
                "␣" buffer-file-name))
  )

(add-hook 'c++-mode-hook (lambda () (c-set-style "stroustrup")
                                    (flymake-mode t)))
```

### 1.2. Makefile. Hash: 7381d22266f4ef5a9a601b80a76a956c

```
check-syntax:
	g++ -Wall -fsyntax-only $(CHK_SOURCES)
```

### 1.3. .vimrc. Hash: da63747b3e94a58450094526d21a9e41

```
syn on
set nocp number ai si ts=4 sts=4 sw=4
ab #i #include
```

### 1.4. Hash generator. Hash: 0d22aecd779fc370b30a2c628aff517c

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

### 1.5. **Solution template.** Hash: c622504b57694424c5afc298785ca147

```
#include <algorithm>
#include <cassert>
#include <cmath>
#include <cstdio>
#include <cstdlib>
#include <cstring>
#include <ctime>
#include <iostream>
#include <map>
#include <queue>
#include <set>
#include <sstream>
#include <string>
#include <utility>
#include <vector>

using namespace std;

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); }

int main() {
}
```

## 2. Graph algorithms

### 2.1. **Tarjan's SCC algorithm.** Hash: 16e646ee186fcff5ed68116af46b0820

```
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 maximum flow 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. **Successive shortest paths mincost maxflow algorithm.** Hash: 1899233cb68a8d5f6e280654146e1747

```
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, -cst, 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 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);
}
```

## 2.4. **Gabow's general matching algorithm.** Hash: a173c8d78c8dfd97bfacfc98e6ef92ea

```
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: 0c94974a1a54f572893029cececcbe23

```
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: a56c42a3dac08d2c5fc932f92468dd03

```
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: 97a42d1ac6750f912c5a06e04636c1db

```
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: 5288cd2ac5d62a97ea1175eec20d0010

```
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;
}

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);
}
```

### 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: 621a339a657d07de8f651d55e13d988b

```
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;
}
```

### 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: 00dd4dbd6779989a64c1e935443a1d80

```
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: 2b14ae1a97e5ff686efb4d7e0e7ca78a

```
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;
}
```

## 4.7. **Closest pair of points.** Hash: d704271ff258aac5dad13bb04cf0cfb6

```
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);
}
```

## 4.8. **Kd-tree.** Hash: de78e67c89c057ba920d2060641a7f48

```
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 ? 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(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: 47db21e0b6328b90025fa4e9c03e3431

```
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(a, c), compy)
         - tree[0].begin();
      posr = upper_bound(tree[0].begin(), tree[0].end(), pt(b, 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. **Karkkainen-Sanders' suffix array algorithm.** Hash: 1e95f2291a17b091e96a770909448e78

```
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+3], tmp[MAXSZ+3];
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 mod0[MAXSZ/3+10];
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(n);
      for(int i = 0; i < sz2; i++) sub[rec[i]] = i+1;
   } else {
      rec.resize(n);
      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, mod0, 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, pmod0 = sz-1, pret = n-1, srec = n%3==1;
   while(prec >= srec && pmod0 >= 0)
      if(rec[prec]%3 == 1 && k_cmp(v[mod0[pmod0]], v[rec[prec]],
                               sub[mod0[pmod0]/3], sub[rec[prec]/3+sz]) ||
        rec[prec]%3 == 2 && k_cmp(v[mod0[pmod0]], v[rec[prec]],
                               v[mod0[pmod0]+1], v[rec[prec]+1],
                               sub[mod0[pmod0]/3+sz], sub[rec[prec]/3+1]))
         rec[pret--] = rec[prec--];
      else
         rec[pret--] = mod0[pmod0--];
   if(pmod0 >= 0) memcpy(&rec[0], mod0, sizeof(int) * (pmod0+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 < (int)s.size() && s[i+h] == s[j+h]) h++;
         ans[prm[i]-1] = h;
         if(h) h--;
      }
   return ans;
}
```

### 6.2. **Morris-Pratt's algorithm.** Hash: 0234dfb6e26b39d35704838d84f1e86e

```
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 = -1; //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;
         k = pi[k];
      }
   }
}
```

### 6.3. **Aho-Corasick's algorithm (UFPE).** Hash: 273f4391174d22898bfe3f2415f95915

```
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
}
```

## 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 | 5.761.455 |

**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. **Polyominoes.** How many free (rotation, reflection), one-sided (rotation) and fixed $n$-ominoes are there?

| n | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 |
|---|---|---|---|---|---|---|---|---|
| free | 2 | 5 | 12 | 35 | 108 | 369 | 1.285 | 4.655 |
| one-sided | 2 | 7 | 18 | 60 | 196 | 704 | 2.500 | 9.189 |
| fixed | 6 | 19 | 63 | 216 | 760 | 2.725 | 9.910 | 36.446 |

7.10. **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.

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

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

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

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

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

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

## 8. Mistakes to avoid and strategy hints

General strategy hints:

(1) Everyone should have read every problem by the end of the second hour of competition.
(2) **In the last hour, only one question should be attempted at once.**
(3) Always use vectors instead of arrays in the single dimensional case. Use vectors instead of arrays in the bidimensional case if the outermost dimension is not too big. This allows for better debugging.

When thinking (before coding):

(1) **Be organized!**
(2) Don't touch the computer unless the solution is done, including implementation details: avoid thinking too much at the computer. **Time spent on paper detailing a solution is time well spent.**
(3) A corollary to the above: **Don't touch the computer if you doubt your idea**.

In case you have no solution ideas:

(1) (Include tricks from Pólya here)

In case of Wrong Answer:

(1) Make sure the algorithm is correct (i.e. **sketch a proof of correctness**) as soon as possible. Take your time and check it carefully. **If you're sure the idea is correct, make sure you don't second-guess it** when checking for bugs, even if an implementation bug is nowhere to be found.
(2) If the code uses vectors (c.f. general strategy hints), add #define _GLIBCXX_DEBUG at the very beginning of the source code and submit it again. A runtime error means out-of-bounds array access or other STL misuse.
(3) Debug on paper, and don't go back to the computer for every bug you find: **check the whole solution at least once more after finding each new bug.**
(4) Think of tricky test cases.
(5) Read the problem again. For every constraint found, check the printed source code.
(6) If you can't find any bug in five minutes, **go to the bathroom**. If you still can't find the bug, **go to another problem and come back to the wrong solution later**. Debugging a single program for too long leads to finding a lot of false bugs and makes it easy to overlook simple mistakes.

In case of Runtime Error:

(1) Check divisions and modulo operations.
(2) Check array indices (both in declarations and in accesses).
(3) Check for infinite recursion.

# ACM ICPC Team Reference - Contents

Team Anuncie Aqui
Universidade Federal de Sergipe

## Contents