ACM ICPC Team Reference 2010 World Finals

Team Anuncie Aqui
Universidade Federal de Sergipe

Contents

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

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

2. Graph algorithms

2.1. **Dijkstra's algorithm.** Hash: c182c6dfdc4334cb79c7721ae6e88a98

```
int dist[MAXV], last_edge[MAXV], d_visited[MAXV];
int prev_edge[MAXE], weight[MAXE], adj[MAXE];
int nedges;
priority_queue<pair<int, int> > d_q;

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

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

```
void 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));

   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]) {
         int w = adj[i], new_dist = dist[v] + weight[i];
         if(new_dist < dist[w])
            d_q.push(make_pair(-(dist[w] = new_dist), w));
      }
   }
}
```

2.2. **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;
}
```

2.3. **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;
}
```

2.4. **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);
}
```

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;
}
```

4. Geometry

4.1. **Point class.** Hash: 10304b3b4f48e06a26b25e66e389a006

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

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

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

   bool operator==(pt p) { return cmp(x, p.x) == 0 && cmp(y, p.y) == 0; }
   bool operator<(pt p) const {
      return cmp(x, p.x) ? cmp(x, p.x) < 0 : cmp(y, p.y) < 0;
   }
   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; };

TYPE norm(pt a) { return a||a; }
TYPE abs(pt a) { return sqrt(a||a); }
```

4.2. **Primitives.** Hash: 1c388bad0037c675ff5bdcb525ce67be

```
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)); }
inline int g_mod(int i, int n) { if(i == n) return 0; return i; }
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);
}

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

4.3. **Convex hull.** Hash: be47d8cd031deff7a3bf32ef8e9ce115

```
pt pivot;

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

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

```
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)); }

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

```
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);
}

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; }
```

```
   vector<pt> ret;

   pivot = pts[0];
   for(int i = 1; i < pts.size(); i++)
      pivot = min(pivot, pts[i]);
   sort(pts.begin(), 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]);
   }

   return ret;
}
```

4.4. **Kd-tree.** Hash: cbc86c28c44bd28429a302f1c538f3b4

```
#define MAXSZ 10000

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

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

   int mid = (right+left)/2;
   nth_element(pts.begin() + left, pts.begin() + mid,
            pts.begin() + right, 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);
}

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

int kd_query(int root, int a, int b, int c, int d, bool x) {
   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, l, r;
   if(x) l = a, r = b;
   else l = c, r = d;

   if(l <= split[root]) ret += kd_query(2*root + 1, a, b, c, d, !x);
   if(split[root] <= r) ret += kd_query(2*root + 2, a, b, c, d, !x);

   return ret;
}
```

4.5. **Range tree.** Hash: 200c0929f48812967e9b1ec62d7a09c0

```
#define MAXSZ 100000

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], -1e9));
      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 && tree[cl][l].y <= tree[cr][r].y))
```

```
         tree[root].push_back(last = tree[cl][l++]);
      else tree[root].push_back(last = tree[cr][r++]);

      while(llink < tree[cl].size() && tree[cl][llink].y < last.y) llink++;
      while(rlink < tree[cr].size() && tree[cr][rlink].y < last.y) rlink++;

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

   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, int a, int b, int c, int d, int pos = -1) {
   if(root == 0 && pos == -1)
      pos = lower_bound(tree[0].begin(), tree[0].end(), c, compy)
         - tree[0].begin();

   int ret = 0;
   if(a <= xs[l] && xs[r] <= b) {
      while(pos < tree[root].size() && tree[root][pos].y <= d)
         ret++, pos++;
      return ret;
   }
   if(pos >= tree[root].size()) return 0;

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

   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;
      }
   }
};
```

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]);
}
```