Team Reference Document

Comenius University

Obsah

1 Ahov-Corasickovej algoritmus a písmenkový strom (trie)

Štandardné použitie: Spravím si prázdny trie, pomocou `insert` doň navkladám reťazce-ihly. Po vložení posledného z nich zavolám `create_automaton` čím predrátam spätné hrany. Následne vyhľadávam v danom haystacku všetky ihly naraz volaním `report_matched_words`.
Ak chceme inú abecedu ako len lowercase písmená, stačí zmeniť prvé dva riadky.

```
const int ALPHABET_SIZE = 26;
int to_index(char c) { return c-'a'; }

struct trie {
    struct __trieNode {
        vector<int> endsHere;
        __trieNode *nextOutput, *son[ALPHABET_SIZE], *higher;
        __trieNode() { nextOutput=higher=NULL; memset(son,0,sizeof(son)); }
    };

    __trieNode *root;
    int __size;

    trie() { root = NULL; __size = 0; }

    void insert(const string &S) {
        __trieNode *kde;
        if (!root) root = new __trieNode();
        kde = root;
        for (unsigned i=0; i<S.size(); ++i) {
            int idx = to_index( S[i] );
            if (! kde->son[idx]) kde->son[idx] = new __trieNode();
            kde = kde->son[idx];
        }
        kde->endsHere.push_back(__size++);
    }

    int find(const string &S) {
        __trieNode *kde = root;
        for (unsigned i=0; i<S.size(); ++i) {
            if (!kde) return -1;
            kde = kde->son[ to_index( S[i] ) ];
        }
        if (!kde || kde->endsHere.empty()) return -1;
        return kde->endsHere[0];
    }

    void create_automaton() {
        if (!root) return;
        queue<__trieNode*> Q;
        Q.push(root);
        __trieNode *superRoot = new __trieNode();
        for (int i=0; i<ALPHABET_SIZE; ++i) superRoot->son[i] = root;
        root->higher = superRoot;
        while (!Q.empty()) {
            __trieNode *kde = Q.front(); Q.pop();
            for (int i=0; i<ALPHABET_SIZE; ++i)
                if (kde->son[i]) {
                    __trieNode *cand = kde->higher;
                    while (! cand->son[i]) cand = cand->higher;
                    kde->son[i]->higher = cand->son[i];
                    if (kde->son[i]->higher->endsHere.empty())  kde->son[i]->nextOutput = kde->son[i]->higher->nextOutput;
                    else                                        kde->son[i]->nextOutput = kde->son[i]->higher;
                    Q.push(kde->son[i]);
                }
        }
    }

    set<int> report_matched_words(const string &S) {
        assert(root->higher); // call createAutomaton first!
        set<int> result;
        __trieNode *kde = root;
        for (unsigned i=0; i<S.size(); ++i) {
            int idx = to_index( S[i] );
            while (! kde->son[idx]) kde = kde->higher;
            kde = kde->son[idx];
            __trieNode *out = kde;
            while (out) {
                result.insert( out->endsHere.begin(), out->endsHere.end() );
                out = out->nextOutput;
            }
        }
        return result;
    }
};
```

2 Biconnected komponenty, artikulácie a mosty

Každá hrana dostane do `bcc` číslo jej biconnected komponentu. Mosty sú komponenty tvorené jednou hranou, artikulácie sú vrcholy vo viac ako jednom komponente.

```
struct bcc {
    struct edge { int dest, bcc, twin; };

    vector< vector<edge> > G;

    vector<int> visited;
    stack<int> vstack, estack;
    int cur_bcc, cur_time;

    bcc(int N) { G.resize(N); }

    int search(int v) {
        int result = visited[v] = ++cur_time;
        for (int i = 0; i < int( G[v].size() ); ++i) {
            if (G[v][i].bcc != -1) continue;
            vstack.push(v); estack.push(i);
            int d = G[v][i].dest, t = G[v][i].twin;
            G[d][t].bcc = -2;
            int r = visited[d] ? visited[d] : search(d);
            if (r >= visited[v]) {
                int a,b;
                do {
                    a=vstack.top(); b=estack.top();
                    vstack.pop(); estack.pop();
                    G[a][b].bcc = cur_bcc;
                    G[ G[a][b].dest ][ G[a][b].twin ].bcc = cur_bcc;
                }
                while (a!=v || b!=i);
                ++cur_bcc;
            }
            result = min(result,r);
        }
        return result;
    }

    void add_edge(int x, int y) {
        // assert( max(x,y) < int(G.size()) );
        edge e1, e2;
        e1.dest=y; e1.bcc=-1; e1.twin=G[y].size();
        e2.dest=x; e2.bcc=-1; e2.twin=G[x].size();
        G[x].push_back(e1);
        G[y].push_back(e2);
    }

    void compute_bcc() {
        int N = G.size();
        cur_bcc = cur_time = 1;
        visited.clear();
        visited.resize(N,0);
        for (int n=0; n<N; ++n) if (!visited[n]) search(n);
    }
};
```

3 Bipartitné grafy: párovanie, vertex cover, independent set

Efektívna a stručná implementácia základnej metódy cez zlepšujúce cesty. Praktické testy: rýchlostné::bachelors.

```
struct MaximumMatching {
    vector< vector<int> > AB, BA;

    MaximumMatching(int A, int B); // parametre: velkosti particii
    void add_edge(int a, int b);
    vector< pair<int,int> > maximum_matching();
    pair< vector<int>, vector<int> > minimum_vertex_cover();  // {podmnozina vlavo,vpravo}
    // maximum independent set == komplement minimum vertex coveru
};

MaximumMatching::MaximumMatching(int A, int B) { AB.resize(A); BA.resize(B); }

void MaximumMatching::add_edge(int a, int b) { AB[a].push_back(b); BA[b].push_back(a); }

vector< pair<int,int> >
MaximumMatching::maximum_matching() {
    int A = AB.size(), B = BA.size();
    vector<int> match(A,-1);
    for (int b=0; b<B; ++b) {
        vector<int> from(A,-1);
        queue<int> Q;
        for (int a : BA[b]) { Q.push(a); from[a]=a; }
        while (!Q.empty()) {
            int a = Q.front(); Q.pop();
            if (match[a] == -1) {
                while (from[a] != a) { match[a] = match[from[a]]; a = from[a]; }
                match[a] = b;
                break;
            } else {
                for (int x : BA[match[a]]) if (from[x]==-1) { Q.push(x); from[x]=a; }
            }
        }
    }
    vector< pair<int,int> > result;
    for (int a=0; a<A; ++a) if (match[a] != -1) result.push_back( { a, match[a] } );
    return result;
}

pair< vector<int>, vector<int> >
MaximumMatching::minimum_vertex_cover() {
    vector< pair<int,int> > matching = maximum_matching();

    int A = AB.size(), B = BA.size();
    vector<int> match(A,-1);
    vector<bool> matched(B,false), visited(A,false);
    queue<int> Q;

    for (auto p : matching) { match[p.first]=p.second; matched[p.second]=true; }
    for (int b=0; b<B; ++b) if (!matched[b])
        for (int a : BA[b]) if (!visited[a]) { visited[a]=true; Q.push(a); }

    while (!Q.empty()) {
        int a = Q.front(); Q.pop();
        for (int x : BA[match[a]]) if (!visited[x]) { Q.push(x); visited[x]=true; }
    }

    vector<int> left_cover, right_cover;
    for (int a=0; a<A; ++a)
        if (visited[a]) left_cover.push_back(a);
        else if (match[a] != -1) right_cover.push_back( match[a] );

    return make_pair( left_cover, right_cover );
}
```

Hopcroft-Karp, dlhší kód ale efektívnejší. Praktické testy: SPOJ::MATCHING, SPOJ::TAXI, SPOJ::MMATCH

```
struct MaximumMatching {
    int A, B;                   // A, B are partition sizes; each is numbered from 0
    vector<int> matching;       // for each b, either -1 or the matching a
    vector< vector<int> > G;
    MaximumMatching(int A, int B) : A(A), B(B) { G.resize(A); }
    void add_edge(int left, int right);
    void compute_matching();
};

inline void MaximumMatching::add_edge(int left, int right) {
```

```
    // assert(A > left); assert(B > right);
    G[left].push_back(right);
}

void MaximumMatching::compute_matching() {
    // greedy matching
    matching.clear(); matching.resize(B,-1);
    for (int a=0; a<A; ++a) for (int b : G[a]) if (matching[b]==-1) { matching[b]=a; break; }

    // improvements
    while (1) {
        vector<bool> done(B,false);
        vector<int> lpred(A,-2), first(B,-1), next, rpred, unmatched, layer, temp;

        for (int b=0; b<B; ++b) if (matching[b]>=0) lpred[matching[b]]=-1;
        for (int a=0; a<A; ++a) if (lpred[a]==-2) layer.push_back(a);

        // repeatedly extend layering structure by another pair of layers
        while (!layer.empty() && unmatched.empty()) {
            temp.clear();
            for (int u : layer) for (int v : G[u]) if (!done[v]) {
                int n = rpred.size(), s = first[v];
                rpred.push_back(u);
                next.push_back(s);
                first[v] = n;
                if (s==-1) temp.push_back(v);
            }
            layer.clear();
            for (int b : temp) {
                done[b] = true;
                if (matching[b]==-1) unmatched.push_back(b); else {
                    layer.push_back( matching[b] );
                    lpred[ matching[b] ] = b;
                }
            }
        }

        // did we finish layering without finding any alternating paths?
        if (unmatched.empty()) return;

        for (int v : unmatched) {
            stack<int> vrchol, hrana, oldhrana;
            vrchol.push(v); hrana.push(first[v]); oldhrana.push(first[v]);
            while (!vrchol.empty()) {
                int kde=vrchol.top(), ako=hrana.top();
                vrchol.pop(); hrana.pop(); oldhrana.pop();
                if (ako == -1) continue;
                vrchol.push(kde); hrana.push(next[ako]); oldhrana.push(ako);
                int kam = rpred[ako], p = lpred[kam];
                lpred[kam] = -1;
                if (p==-1) continue;
                if (p==-2) {
                    while (!vrchol.empty()) {
                        matching[ vrchol.top() ] = rpred[oldhrana.top()];
                        vrchol.pop(); oldhrana.pop();
                    }
                    break;
                }
                vrchol.push(p); hrana.push(first[p]); oldhrana.push(first[p]);
            }
        }
    }
}
```

4 Diskrétny logaritmus, extended GCD

Diskrétny logaritmus: praktické testy: UVa 10225

```
/*
    extended GCD
    input: A,B >= 0
    output: pair(d, pair(u,v)), where d == __gcd(A,B) == u*A+v*B
*/
pair< long long, pair<long long,long long> > EGCD(long long A, long long B) {
    if (B==0) return { A, {1,0} };
    auto tmp = EGCD(B,A%B);
    return { tmp.first, { tmp.second.second, tmp.second.first - tmp.second.second * (A/B) } };
}


/*
    MODEXP expects: 0<number<2^31, 0<=power<2^63, 0<modulus<2^31
    MODEXP returns: (number^power) % modulus
*/
long long MODEXP(long long number, long long power, long long modulus) {
    if (power==0) return 1LL % modulus;
    if (power==1) return number % modulus;
    long long tmp = MODEXP(number,power/2,modulus);
    tmp = (tmp*tmp) % modulus;
    if (power&1) tmp = (tmp*number) % modulus;
    return tmp;
}

/*
    O( \sqrt{P} \log P )    discrete logarithm algorithm by Shanks

    parameters: A, B, P
    expects: P is a prime, 1 < A < P < 2^31

    returns: one possible \log_A B (mod P)     or -1, if none
*/
long long DLOG (long long A, long long B, long long P) {
    long long M = (long long)ceil(sqrt(P-1.0));

    vector< pair<long long, int> > P1, P2;
    long long pom = MODEXP(A,M,P);

    P1.push_back(make_pair(1,0));
    for (int i=1; i<M; i++) P1.push_back(make_pair( (P1[i-1].first * pom)%P, i));
    sort(P1.begin(), P1.end());

    long long Ainv = MODEXP(A,P-2,P);
    P2.push_back(make_pair(B,0));
    for (int i=1; i<M; i++) P2.push_back(make_pair( (P2[i-1].first * Ainv)%P, i));
    sort(P2.begin(), P2.end());

    int i,j;
    for (i=0, j=0; P1[i].first != P2[j].first; ) {
        if (P1[i].first < P2[j].first) i++; else j++;
        if ( i==M || j==M ) return -1;
    }
    return ( M * P1[i].second + P2[j].second ) % (P-1);
}
```

5 Fenwickov („fínsky“) strom a iná bitwise mágia

Iterovanie cez všetky podmnožiny danej množiny `super`:

```
for ( int sub=0; sub=sub-super&super; )
```

(Užitočné na ľahké nakódenie skúšania všetkých možností pre $A \subseteq B \subseteq \{0, \ldots, n-1\}$ v čase 3^n.)

Nasledujú Fenwickove stromy. Dajú sa do nich vkladať frekvencie pre prvky s číslami od 1 po `maxval` vrátane. Na správne fungovanie `find` (v 1D verzii) treba aby všetky frekvencie boli nezáporné.

```
struct Fenwick1D {
    int size;
    vector<int> T;

    Fenwick1D(int maxval) {
        size = 1;
        while (size < maxval) size <<= 1;
        T.clear();
        T.resize(size+1,0);
    }

    void update(int x, int delta) { // assumes 1 <= x <= init_maxval
        while (x <= size) { T[x] += delta; x += x & -x; }
    }

    int sum(int x1, int x2) { // sum in the closed interval [x1,x2]
        int res=0;
        --x1;
        while (x2) { res += T[x2]; x2 -= x2 & -x2; }
        while (x1) { res -= T[x1]; x1 -= x1 & -x1; }
        return res;
    }

    int find(int sum) { // largest z such that sum( [1,z] ) <= sum
        int idx = 0, bitMask = size;
        while (bitMask && (idx < size)) {
            int tIdx = idx + bitMask;
            if (sum >= T[tIdx]) { idx=tIdx; sum -= T[tIdx]; }
            bitMask >>= 1;
        }
        return idx;
    }
};

struct Fenwick2D {
    int sizer, sizec;
    vector< vector<int> > T;

    Fenwick2D(int maxr, int maxc) {
        sizer = 1; while (sizer < maxr) sizer <<= 1;
        sizec = 1; while (sizec < maxc) sizec <<= 1;
        T.resize( sizer+1, vector<int>(sizec+1,0) );
    }

    void update(int x, int y, int delta) {
        while (x <= sizer) {
            int y1 = y;
            while (y1 <= sizec) { T[x][y1] += delta; y1 += y1 & -y1; }
            x += x & -x;
        }
    }

    int sum(int x1, int _y1, int x2, int _y2) {
        int res=0;
        x1--;
        while (x2) {
            int y1=_y1-1, y2=_y2;
            while (y2) { res += T[x2][y2]; y2 -= y2 & -y2; }
            while (y1) { res -= T[x2][y1]; y1 -= y1 & -y1; }
            x2 -= x2 & -x2;
        }
        while (x1) {
            int y1=_y1-1, y2=_y2;
            while (y2) { res -= T[x1][y2]; y2 -= y2 & -y2; }
            while (y1) { res += T[x1][y1]; y1 -= y1 & -y1; }
            x1 -= x1 & -x1;
        }
        return res;
    }
};
```

6 NTT na konvolúcie

Viď komentár v kóde. Praktické testy rychlostne::zlystvorec.

```
// NTT: Number-theoretic transform -- to iste ako Fourierka, ale v celych cislach s modulom
// MAXNTH_ROOT je cislo ktoreho vsetky mocniny su rozne, az potom (MAXNTH_ROOT^MAXN) mod MOD == 1
// takze sa da pouzivat namiesto MAXN-tej komplexnej odmocniny z 1
//
// ak treba vacsi rozsah a mame 128-bitove inty, tak sa da
// {unsigned, unsigned long long, MAXN, MOD, MAXNTH_ROOT} -> {__uint64_t, __uint128_t, 1ULL << 55, 5*MAXN+1, 3}
// to je ale asi 4x pomalsie

const unsigned MAXN = 1 << 25;
const unsigned MOD = 125 * MAXN + 1;
const unsigned MAXNTH_ROOT = 199;

inline unsigned add     (unsigned a, unsigned b) { return ((unsigned long long) a + b) % MOD; }
inline unsigned subtract(unsigned a, unsigned b) { return ((unsigned long long) a + MOD - b) % MOD; }
inline unsigned multiply(unsigned a, unsigned b) { return ((unsigned long long) a * b) % MOD; }

inline unsigned power   (unsigned a, unsigned b) {
    unsigned long long r = 1;
    unsigned long long m = a;
    while (b > 0) {
        if (b % 2 == 1) r = r * m % MOD;
        m = m * m % MOD;
        b /= 2;
    }
    return r;
}

inline unsigned inverse(unsigned a) { return power(a, MOD - 2); }

void NTT(vector<unsigned>& A, unsigned root) {
    int n = A.size();
    if (n <= 1) return;

    vector<unsigned> B(n/2), C(n/2);
    for (int i=0; i<n/2; ++i) { B[i] = A[2*i]; C[i] = A[2*i+1]; }
    unsigned root_square = multiply(root, root);
    NTT(B, root_square);
    NTT(C, root_square);
    unsigned root_i = 1;
    for (int i=0; i<n/2; ++i) {
        A[i] = add(B[i], multiply(root_i, C[i]));
        A[i + n / 2] = subtract(B[i], multiply(root_i, C[i]));
        root_i = multiply(root_i, root);
    }
}

void NTT_inverse(vector<unsigned>& A, unsigned root) {
    NTT(A, inverse(root));
    unsigned n_inverse = inverse( A.size() );
    for (unsigned i=0; i<A.size(); ++i) A[i] = multiply(A[i], n_inverse);
}

// NTT_multiply vynasobi (sum_i A[i] * x^i) * (sum_j B[j] * x^j)
vector<unsigned> NTT_multiply(const vector<unsigned> &A, const vector<unsigned> &B) {
    int N = 1;
    while (N < int(A.size() + B.size()) + 47) N *= 2;

    vector<unsigned> CA = A; CA.resize(N,0);
    vector<unsigned> CB = B; CB.resize(N,0);

    unsigned root = power(MAXNTH_ROOT, MAXN / N);
    NTT(CA,root);
    NTT(CB,root);
    for (int n=0; n<N; ++n) CA[n] = multiply( CA[n], CB[n] );
    NTT_inverse(CA,root);

    while (!CA.empty() && CA.back()==0) CA.pop_back();
    return CA;
}
```

7 Geometria v rovine

```
/*
 * geometry version 1.0.2
 *
 * COORD_TYPE may be { int, long long, double, long double }.
 * at least double precision required (e.g., for inputs up to 10^9 use long long)
 *
 * the following stuff does not work properly if COORD_TYPE is an integer type:
 * normalize, center_of_mass, intersect_*, distance_*, rotate, circumcircle_center
 *
 * if a poly stores a polygon, DO NOT repeat vertex 0 as vertex N
 *
 */

typedef double COORD_TYPE;
typedef complex<COORD_TYPE> point;
typedef vector<point> poly;

#define EPSILON (1e-7) // epsilon used for computations involving doubles ; lower to 1e-9 for %Lf

// safe comparison with 0: is_zero, is_negative, is_positive, signum
#define INTEGER_COMPARISONS(type) \
    bool is_negative(type x) { return x<0; } \
    bool is_zero(type x) { return x==0; } \
    bool is_positive(type x) { return x>0; }
INTEGER_COMPARISONS(int)
INTEGER_COMPARISONS(long long)

#define FLOAT_COMPARISONS(type) \
    bool is_negative(type x) { return x < -EPSILON; } \
    bool is_zero(type x) { return abs(x) <= EPSILON; } \
    bool is_positive(type x) { return x > EPSILON; }
FLOAT_COMPARISONS(double)
FLOAT_COMPARISONS(long double)

template<class T> int signum(const T &A) { if (is_zero(A)) return 0; if (is_negative(A)) return -1; return 1; }

// safe equality test for points
bool are_equal(const point &A, const point &B) { return is_zero(real(B)-real(A)) && is_zero(imag(B)-imag(A)); }

// cross-product, dot_product, square_size (=dot_product(A,A)) for 2D vectors
COORD_TYPE square_size  (const point &A)                 { return real(A) * real(A) + imag(A) * imag(A); }
COORD_TYPE dot_product  (const point &A, const point &B) { return real(A) * real(B) + imag(A) * imag(B); }
COORD_TYPE cross_product(const point &A, const point &B) { return real(A) * imag(B) - real(B) * imag(A); }

// size, normalize for 2D real vectors
COORD_TYPE size(const point &A) { return sqrt(real(A) * real(A) + imag(A) * imag(A)); }
void normalize(point &A) { COORD_TYPE Asize = size(A); A *= (1/Asize); }

// safe colinearity and orientation tests: colinear, clockwise, counterclockwise
bool colinear(const point &A, const point &B, const point &C) { return is_zero( cross_product( B-A, C-A )); }
bool colinear(const point &B, const point &C)                 { return is_zero( cross_product( B,   C ));   }
bool clockwise(const point &A, const point &B, const point &C) { return is_negative( cross_product( B-A, C-A )); }
bool clockwise(const point &B, const point &C)                 { return is_negative( cross_product( B,   C ));   }
bool counterclockwise(const point &A, const point &B, const point &C) { return is_positive( cross_product( B-A, C-A )); }
bool counterclockwise(const point &B, const point &C)                 { return is_positive( cross_product( B,   C ));   }

// polygon area: twice_signed_poly_area, poly_area
COORD_TYPE twice_signed_poly_area(const poly &V) {
    COORD_TYPE res = 0;
    for (unsigned i=0; i<V.size(); i++) res += cross_product( V[i], V[(i+1)%V.size()] );
    return res;
}

long double poly_area(const poly &V) { return abs(0.5 * twice_signed_poly_area(V)); }

// compute the center of mass of a polygon: center_of_mass
point center_of_mass(const poly &V ) {
    point sum(0,0);
    for (unsigned i=0; i<V.size(); i++)
        sum += cross_product( V[i], V[(i+1)%V.size()] ) * ( V[i] + V[(i+1)%V.size()] );
    return sum * 1.0 / (3.0 * twice_signed_poly_area(V));
}

// safe comparison function compare_XY
bool compare_XY(const point &A, const point &B) {
    if (!is_zero(real(A)-real(B))) return (is_negative(real(A)-real(B))); else return (is_negative(imag(A)-imag(B)));
}

// compute a CCW convex hull with no unnecessary points: convex_hull
poly convex_hull(poly P) {
    int n = P.size(), k = 0;
    if (n <= 1) return P;
```

```
    sort( P.begin(), P.end(), compare_XY );
    poly H(2*n);
    for (int i = 0; i < n; i++) {
        while (k >= 2 && !counterclockwise(H[k-2], H[k-1], P[i])) --k;
        H[k++] = P[i];
    }
    for (int i = n-2, t = k+1; i >= 0; i--) {
        while (k >= t && !counterclockwise(H[k-2], H[k-1], P[i])) --k;
        H[k++] = P[i];
    }
    H.resize(k-1);
    if (H.size()==2u && are_equal(H[0],H[1])) H.pop_back(); // if all points were identical
    return H;
}

// safe test whether a point C \in [A,B], C \in (A,B): is_on_segment, is_inside_segment
bool is_on_segment(const point &A, const point &B, const point &C) {
    if (!colinear(A,B,C)) return false; else return ! is_positive( dot_product(A-C,B-C) );
}

bool is_inside_segment(const point &A, const point &B, const point &C) {
    if (!colinear(A,B,C)) return false; else return is_negative( dot_product(A-C,B-C) );
}

// winding number of a poly around a point (not on its boundary): winding_number(pt,poly)
int winding_number( const point &bod, const poly &V ) {
    int wn = 0;
    for (unsigned i=0; i<V.size(); i++) {
        if (! is_positive( imag(V[i]) - imag(bod) )) {
            if (is_positive( imag(V[(i+1)%V.size()]) - imag(bod) ))
                if (counterclockwise( V[i], V[(i+1)%V.size()], bod )) ++wn;
        } else {
            if (! is_positive( imag(V[(i+1)%V.size()]) - imag(bod) ))
                if (clockwise( V[i], V[(i+1)%V.size()], bod )) --wn;
        }
    }
    return wn;
}

// test whether a point is inside a polygon: is_inside
int is_inside( const point &bod, const poly &V ) {
    for (unsigned i=0; i<V.size(); i++) if (is_on_segment(V[i],V[(i+1)%V.size()],bod)) return true;
    return winding_number(bod,V) != 0;
}

// test whether a polygon is convex (assuming that the input is a valid polygon!): is_poly_convex
bool is_poly_convex(const poly &V) {
    int cw=0, ccw=0;
    unsigned N = V.size();
    for (unsigned i=0; i<V.size(); i++) if (clockwise( V[i], V[(i+1)%N], V[(i+2)%N] ))        { cw=1; break; }
    for (unsigned i=0; i<V.size(); i++) if (counterclockwise( V[i], V[(i+1)%N], V[(i+2)%N] )) { ccw=1; break; }
    return (! (cw && ccw));
}

// test whether a polygon is given in clockwise order: is_poly_clockwise
bool is_poly_clockwise(const poly &V) { return is_negative(twice_signed_poly_area(V)); }

// intersect lines (A,B) and (C,D), return 2 pts if identical: intersect_line_line
poly intersect_line_line(const point &A, const point &B, const point &C, const point &D) {
    point U = B-A, V = D-C;
    if (colinear(U,V)) { if (colinear(U,C-A)) return {A,B}; else return {}; } // identical or parallel
    COORD_TYPE k = ( cross_product(C,V) - cross_product(A,V) ) / cross_product(U,V);
    return { A+k*U }; // one intersection point
}

// intersect segments [A,B] and [C,D], may return endpoints of a segment: intersect_segment_segment
poly intersect_segment_segment(const point &A, const point &B, const point &C, const point &D) {
    point U = B-A, V = D-C, W = C-A, X = D-A;
    if (colinear(U,V)) { // parallel
        // check for degenerate cases
        if (are_equal(A,B)) { if (is_on_segment(C,D,A)) return {A}; }
        if (are_equal(C,D)) { if (is_on_segment(A,B,C)) return {C}; }
        // two parallel segments
        if (!colinear(U,W)) return {}; // not on the same line

        // A, B, C, D all lie on the same non-degenerate line; if A=0 and B=1, what are C and D?
        COORD_TYPE a = 0, b = 1;
        COORD_TYPE c = dot_product(U,W) / square_size(U);
        COORD_TYPE d = dot_product(U,X) / square_size(U);
        COORD_TYPE lo = max(a,min(c,d)), hi = min(b,max(c,d));

        if (is_negative(hi-lo)) return {}; // no intersection
        if (is_zero(hi-lo)) return { A+U*lo }; // one point
        return { A+U*lo, A+U*hi };
    }
    // not parallel, at most one intersection point
```

```
    COORD_TYPE k = ( cross_product(C,V) - cross_product(A,V) ) / cross_product(U,V);
    point cand = A + k*U;
    if (is_on_segment(A,B,cand) && is_on_segment(C,D,cand)) return {cand}; else return {};
}

// intersect a convex poly and a halfplane to the left of [A,B): intersect_cpoly_halfplane
poly intersect_cpoly_halfplane(const poly &V, const point &A, const point &B) {
    int N = V.size();
    poly res;
    if (N == 0) return res;
    if (N == 1) { if (! clockwise(A,B,V[0]) ) return V; else return res; }

    for (int i=0; i<N; i++) {
        if (! clockwise(A,B,V[i])) res.push_back(V[i]);
        bool intersects = false;
        if (counterclockwise(A,B,V[i])) if (clockwise(A,B, V[(i+1)%N] )) intersects = true;
        if (clockwise(A,B,V[i])) if (counterclockwise(A,B, V[(i+1)%N] )) intersects = true;
        if (intersects) {
            poly tmp = intersect_line_line(A,B,V[i], V[(i+1)%N] );
            res.push_back( tmp[0] ); // vieme, ze ma len jeden bod
        }
    }
    return res;
}

// intersect two convex polygons SLOWLY -- in O(n^2). Result is CCW. intersect_cpoly_cpoly
poly intersect_cpoly_cpoly(poly V1, poly V2) {
    if (is_poly_clockwise(V1)) reverse(V1.begin(),V1.end());
    if (is_poly_clockwise(V2)) reverse(V2.begin(),V2.end());
    for (unsigned i=0; i<V2.size(); i++) V1 = intersect_cpoly_halfplane(V1,V2[i],V2[(i+1) % V2.size() ]);
    return V1;
}

// intersect circles (C1,r1) and (C2,r2), 3 pts returned if \equiv: intersect_circle_circle
poly intersect_circle_circle(const point &C1, COORD_TYPE r1, const point &C2, COORD_TYPE r2) {
    if (are_equal(C1,C2)) {
        if (is_zero(r1) && is_zero(r2)) return {C1}; // 2x the same point
        if (is_zero(r1-r2)) return {C1,C1,C1};       // identical circles -- return 3 points
        return {};                                   // different radii -- no intersection
    }

    COORD_TYPE d = sqrt(square_size(C2-C1));
    // check for no intersection
    if (is_positive(d-r1-r2) || is_positive(r1-r2-d) || is_positive(r2-r1-d)) return {};
    // check for a single intersection
    if (is_zero(d-r1-r2)) return { (1.0/d) * ( r2*C1 + r1*C2 ) };
    if (is_zero(r1-r2-d)) return { C1 + (r1/d) * (C2-C1) };
    if (is_zero(r2-r1-d)) return { C2 + (r2/d) * (C1-C2) };
    // general case: compute x and y offset of the intersections
    COORD_TYPE x = ( d*d - r2*r2 + r1*r1 ) / (2*d);
    COORD_TYPE y = sqrt( 4*d*d*r1*r1 - ( d*d - r2*r2 + r1*r1 )*( d*d - r2*r2 + r1*r1 ) ) / (2*d);
    // I = (C1,C2) \cap chord, N = normal vector
    point I = (1.0/d) * ( (d-x)*C1 + x*C2 );
    point N( imag(C2-C1), -real(C2-C1) );
    COORD_TYPE Nsize = sqrt(square_size(N));
    N = N * (1/Nsize);
    // compute and return the points in lexicographic order
    point I1 = I + y*N;
    point I2 = I - y*N;
    if (is_positive(real(I1)-real(I2))) swap(I1,I2);
    if (is_zero(real(I1)-real(I2))) if (is_positive(imag(I1)-imag(I2))) swap(I1,I2);
    return {I1,I2};
}

// intersects the circle (center C, radius r) and the infinite line through A and B:
poly intersect_circle_line(const point &C, double r, const point &A, const point &B) {
    point d = B-A, f = A-C;
    double a = dot_product(d,d), b=2*dot_product(f,d), c=dot_product(f,f)-r*r;
    double discriminant = b*b-4*a*c;
    if ( discriminant < 0 ) return {};
    discriminant = sqrt( discriminant );
    if (is_zero(discriminant)) { double t = (-b)/(2*a); return { A+t*d }; }
    double t1 = (-b - discriminant)/(2*a), t2 = (-b + discriminant)/(2*a);
    // pre segment by tu sli testy ci t1, t2 \in [0,1]
    return { A+t1*d, A+t2*d };
}

// compute distance: point A, line (B,C): distance_point_line
COORD_TYPE distance_point_line(const point &A, const point &B, const point &C) {
    point N ( -imag(B-C), real(B-C) );
    COORD_TYPE tmp = dot_product(N,B-A);
    return abs( tmp / size(N) );
}

// compute distance: point A, segment [B,C]: distance_point_segment
COORD_TYPE distance_point_segment(const point &A, const point &B, const point &C) {
```

```
    COORD_TYPE res = min( size(B-A), size(C-A) );

    point N ( -imag(B-C), real(B-C) );
    normalize(N);
    COORD_TYPE tmp = dot_product(N,B-A);
    point paeta = A + tmp*N;
    if (is_on_segment(B,C,paeta)) res = min(res,abs(tmp));
    return res;
}

// compute distance: line (A,B), line (C,D): distance_line_line
COORD_TYPE distance_line_line(const point &A, const point &B, const point &C, const point &D) {
    if (!colinear(B-A,D-C)) return 0;
    return distance_point_line(A,C,D);
}

// compute distance: segment [A,B], line (C,D): distance_segment_line
COORD_TYPE distance_segment_line(const point &A, const point &B, const point &C, const point &D) {
    if (! is_positive( cross_product(D-C,A-C) * cross_product(D-C,B-C) )) return 0; // they intersect
    return min( distance_point_line(A,C,D), distance_point_line(B,C,D) );
}

// compute distance: segment [A,B], segment [C,D]: distance_segment_segment
COORD_TYPE distance_segment_segment(const point &A, const point &B, const point &C, const point &D) {
    // TODO: remove the dependency on intersect_segment_segment (matters for manual retyping)
    // note: remove this test if guaranteed that segments are disjoint
    poly isect = intersect_segment_segment(A,B,C,D);
    if (isect.size()) return 0; // they intersect

    COORD_TYPE res =  distance_point_segment(A,C,D);
    res = min(res, distance_point_segment(B,C,D));
    res = min(res, distance_point_segment(C,A,B));
    res = min(res, distance_point_segment(D,A,B));
    return res;
}

// circumcircle center for three points A,B,C (cannot be colinear):
point circumcircle_center(const point &A, const point &B, const point &C) {
    COORD_TYPE a = 0, bx = 0, by = 0;
    a += real(A) * imag(B) + real(B) * imag(C) + real(C) * imag(A);
    a -= real(B) * imag(A) + real(C) * imag(B) + real(A) * imag(C);
    // assert( ! is_zero(a) );
    bx += square_size(A) * imag(B) + square_size(B) * imag(C) + square_size(C) * imag(A);
    bx -= square_size(B) * imag(A) + square_size(C) * imag(B) + square_size(A) * imag(C);
    by -= square_size(A) * real(B) + square_size(B) * real(C) + square_size(C) * real(A);
    by += square_size(B) * real(A) + square_size(C) * real(B) + square_size(A) * real(C);
    return point ( bx / (2*a) , by / (2*a) );
}

// rotate_point(point,center,CCW angle in radians):
point rotate_point(const point &bod, const point &stred, COORD_TYPE uhol) {
    point mul(cos(uhol),sin(uhol));
    return ((bod-stred)*mul)+stred;
}

// rotate_poly(poly,center,CCW angle in radians):
poly rotate_poly(poly V, const point &stred, COORD_TYPE uhol) {
    for (unsigned i=0; i<V.size(); i++) V[i] = rotate_point(V[i],stred,uhol);
    return V;
}

// angle in radians on the left side of B in polyline A->B->C: left_side_angle
long double left_side_angle(const point &A, const point &B, const point &C) {
    long double u = atan2( imag(A-B), real(A-B) ) - atan2( imag(C-B), real(C-B) );
    if (u < 0) u += 2*M_PI;
    if (u >= 2*M_PI) u -= 2*M_PI;
    return u;
}
```

8 Maximum flow

Zbastardená verzia Dinicovho algoritmu – dostatočne efektívna a dostatočne stručná. Praktické testy: SPOJ::FASTFLOW, SPOJ::POTHOLE, SPOJ::NETADMIN, Timus::1736

```
template<class T> struct MaxFlow {
    struct edge {
        int source, destination;
        T capacity, residue;
        edge(int s, int d, T cap, T res) : source(s), destination(d), capacity(cap), residue(res) { }
    };

    vector< vector<int> > V;
    vector<edge> E;

    MaxFlow(int N) { V.resize(N); }
    void add_arc(int source, int destination, T capacity);
    T get_flow(int source, int sink);
};

template<class T> void MaxFlow<T>::add_arc(int source, int destination, T capacity) {
    // assert( max(source,destination) < int( V.size() ) );
    int e = E.size();
    V[source].push_back( e );
    V[destination].push_back( e+1 );
    E.push_back( edge( source, destination, capacity, capacity ) );
    E.push_back( edge( destination, source, capacity, 0 ) );
}

template<class T> T MaxFlow<T>::get_flow(int source, int sink) {
    // assert( max(source,sink) < int( V.size() ) );

    T flowSize = 0;
    int N = V.size();

    while (1) { // use BFS to find augmenting paths
        vector<int> from(N,-1);
        queue<int> Q;
        Q.push(source);
        from[source] = -2;

        while (!Q.empty()) {
            int where = Q.front(); Q.pop();
            for (int e : V[where]) {
                int dest = E[e].destination;  if (from[dest] != -1) continue;
                T res    = E[e].residue;      if (res == 0) continue;
                from[dest] = e;
                Q.push(dest);
                if (dest == sink) break;
            }
            if (from[sink] >= 0) break;
        }

        if (from[sink]==-1) return flowSize; // if there is no path, we are done

        // construct a maximum set of augmenting paths in the graph given by from[]
        for (int e : V[sink]) {
            int where = E[e].destination;         if (from[where]==-1) continue; // no path leads here
            T res = E[e].capacity - E[e].residue; if (res == 0) continue; // can't push anything more

            // walk the path and determine the delta
            T canPush = res;
            int curr = where;
            while (1) {
                if (from[curr] == -2) break;
                canPush = min( canPush, E[ from[curr] ].residue );
                curr    = E[ from[curr] ].source;
            }

            // walk the path again and update capacities
            flowSize       += canPush;
            E[e  ].residue += canPush;
            E[e^1].residue -= canPush;
            curr = where;
            while (1) {
                if (from[curr] == -2) break;
                E[ from[curr]   ].residue -= canPush;
                E[ from[curr]^1 ].residue += canPush;
                curr = E[ from[curr] ].source;
            }
        }
    }
}
```

9 Minimum cost maximum flow

Klasický algoritmus hľadajúci najlacnejšiu zlepšujúcu cestu, vylepšený tým, že máme potenciály vrcholov a tým pádom môžeme použiť Dijkstru namiesto Bellmana-Forda.
Praktické testy: TC::PeopleYouMayKnow

```
template<class T, class U>
struct MincostMaxflow {
    struct edge {
        int source, destination;
        T capacity, residue;
        U cost;
        edge(int s, int d, T cap, T res, U cost)
            : source(s), destination(d), capacity(cap), residue(res), cost(cost) { }
    };

    MincostMaxflow(int N);
    vector< vector<int> > V;
    vector<edge> E;
    void add_arc(int source, int destination, T capacity, U cost);
    pair<T,U> get_flow(int source, int sink);
};

template<class T, class U>
MincostMaxflow<T,U>::MincostMaxflow(int N) { V.resize(N); }

template<class T, class U>
void MincostMaxflow<T,U>::add_arc(int source, int destination, T capacity, U cost) {
    // assert(source < int(V.size())); assert(destination < int(V.size()));
    int e = E.size();
    V[source].push_back( e );
    V[destination].push_back( e+1 );
    E.push_back( edge( source, destination, capacity, capacity, cost));
    E.push_back( edge( destination, source, capacity, 0, -cost));
}

template<class T, class U>
pair<T,U> MincostMaxflow<T,U>::get_flow(int source, int sink) {
    // assert(source < int(V.size())); assert(sink < int(V.size()));
    int N = V.size(), M = E.size();
    T flowSize = 0;
    U flowCost = 0;

    // automagically initialize infinities and epsilon
    // explicit initialization for ints: infinity = 1<<29, ignore epsilon (epsilon = 0)
    // explicit initialization for doubles: infinity = 1e30, epsilon = 1e-9
    T Tinfinity = 1; while (2*Tinfinity > Tinfinity) Tinfinity *= 2;
    U Uinfinity = 1; while (2*Uinfinity > Uinfinity) Uinfinity *= 2;
    U Uepsilon = 1; for (int i=0; i<30; i++) Uepsilon /= 2;

    vector<T> flow(M,0);
    vector<U> potential(N,0);
    while (1) {
        // use dijkstra to find an augmenting path
        vector<int> from(N,-1);
        vector<U> dist(N,Uinfinity);

        priority_queue< pair<U,int>, vector<pair<U,int> >, greater<pair<U,int> > > Q;
        Q.push( make_pair(0,source) );
        from[source] = -2;
        dist[source] = 0;

        while (!Q.empty()) {
            U howFar  = Q.top().first;
            int where = Q.top().second;
            Q.pop();
            if (dist[where] < howFar) continue;

            for (int i=0; i < int(V[where].size()); i++) {
                if (E[ V[where][i] ].residue == 0) continue;
                int dest = E[ V[where][i] ].destination;
                U cost = E[ V[where][i] ].cost;
                if (dist[dest] > dist[where] + potential[where] - potential[dest] + cost + Uepsilon) {
                    dist[dest] = dist[where] + potential[where] - potential[dest] + cost;
                    from[dest] = V[where][i];
                    Q.push( make_pair( dist[dest],dest ));
                }
            }
        }

        // update vertex potentials
        for (int i=0; i<N; i++) {
            if (dist[i] == Uinfinity) potential[i] = Uinfinity;
            else if (potential[i] < Uinfinity) potential[i] += dist[i];
```

```
        }

        // if there is no path, we are done
        if (from[sink] == -1) return make_pair(flowSize,flowCost);

        // construct an augmenting path
        T canPush = Tinfinity;
        int where = sink;

        while (1) {
            if (from[where] == -2) break;
            canPush = min(canPush, E[ from[where] ].residue );
            where = E[ from[where] ].source;
        }

        // update along the path
        where = sink;
        while (1) {
            if (from[where] == -2) break;
            E[ from[where] ].residue -= canPush;
            E[ from[where]^1 ].residue += canPush;
            flowCost += canPush * E[ from[where] ].cost;
            where = E[ from[where] ].source;
        }
        flowSize += canPush;
    }
    return make_pair(-1,47);
}
```

10 Silno súvislé komponenty

Praktické testy: UVa 10731, liahen::bottom.

```
struct scc {
    vector< vector<int> > G;

    stack<int> S;
    vector<int> SCC, cislo, najmenej;
    vector<bool> spracuvam;
    int cas, nSCC;

    void hladaj(int v) {
        cislo[v] = najmenej[v] = cas++;
        S.push(v); spracuvam[v] = true;
        for (int i=0; i<int( G[v].size() ); i++) {
            int kam = G[v][i];
            if (cislo[kam]==-1) { hladaj(kam); najmenej[v] = min( najmenej[v], najmenej[kam] ); }
            else if (spracuvam[kam]) najmenej[v] = min( najmenej[v], najmenej[kam] );
        }
        if (najmenej[v] == cislo[v]) {
            while (1) { int k = S.top(); S.pop(); spracuvam[k]=false; SCC[k]=nSCC; if (k==v) break; }
            nSCC++;
        }
    }

    void compute_SCCs() {
        int N = G.size();
        SCC.clear();       SCC.resize(N,-1);
        cislo.clear();     cislo.resize(N,-1);
        najmenej.clear();  najmenej.resize(N,-1);
        spracuvam.clear(); spracuvam.resize(N,false);
        cas = 0;
        nSCC = 0;
        for (int i=0; i<N; i++) if (SCC[i]==-1) hladaj(i);
    }
};
```

11 SPRP test prvočíselnosti, Pollard rho faktorizácia, Eratostenovo sito

Faktorizáciu cez Pollardovu ró metódu aj SPRP test prvočíselnosti je vhodné používať pre čísla od 10^9 do 10^{18}. Praktické testy: SPOJ::DIVSUM2, SPOJ::PAGAIN (len SPRP časť), SPOJ::FACT0, Timus::1854.

```
typedef unsigned long long ull;

// given 0 <= a,b,n < 2^63, computes (a*b)%n without overflow
inline ull safe_mul(ull a, ull b, ull n) { return (((__int128_t)a)*b)%n; }

// if the above does not work (128-bit type not present), use the version below
// note: subtractions are significantly faster than modulo
// ull res=0;
// while (b) {
//     if (b&1) { res += a; if (res>=n) res-=n; }
//     b >>= 1; a <<= 1; if (a>=n) a-=n;
// }
// return res;

// given 0 <= a,b,n < 2^63, computes (a^b)%n without overflow
inline ull safe_exp(ull a, ull b, ull n) {
    ull res = 1%n;
    for (ull pw=1; pw <= b; pw <<= 1) {
        if (b & pw) res = safe_mul(res,a,n);
        a = safe_mul(a,a,n);
    }
    return res;
}

// given 2 <= n,a < 2^63, n odd, check whether n is a-SPRP
inline bool is_SPRP(ull n, ull a) {
    if (n%a==0) return false;
    ull d=n-1;
    int s=0;
    while (d%2==0) { s++; d/=2; }
    ull cur = safe_exp(a,d,n);
    if (cur == 1) return true;
    for (int r=0; r<s; r++) {
        if (cur == n-1) return true;
        cur = safe_mul(cur,cur,n);
    }
    return false;
}

// given 0 <= n < 2^63, check whether it is prime
unsigned __small_tests[]    = {2,3,5,7},
         __medium_tests[]   = {2,3,7,61,24251},
         __large_tests[]    = {2,325,9375,28178,450775,9780504,1795265022};
ull      __small_threshold  = 1ULL<<32,
         __medium_threshold = 1ULL<<45;

inline bool is_prime(ull N) {
    if (N < 2) return false;
    unsigned *tests=__large_tests; int test_count=7;
    if (N <= __medium_threshold) { tests=__medium_tests; test_count=5; }
    if (N <= __small_threshold) { tests=__small_tests; test_count=4; }
    for (int i=0; i<test_count; ++i) if (N==tests[i]) return true;
    for (int i=0; i<test_count; ++i) if ( ! is_SPRP(N,tests[i])) return false;
    return true;
}

// Pollard's rho algorithm: given a composite n < 2^63, find its prime factor
// caution: "a" and internal states must be signed!
ull pollard_factor(long long n, long long a, int x0) {
    long long x=x0, y=x0, q=1;
    for (int i=1, j=1; ; i++) {
        x = (a+safe_mul(x,x,n)); if (x>=n) x-=n; if (x<0) x+=n;
        y = (a+safe_mul(y,y,n)); if (y>=n) y-=n; if (y<0) y+=n;
        y = (a+safe_mul(y,y,n)); if (y>=n) y-=n; if (y<0) y+=n;
        q = safe_mul(q,abs(x-y),n);
        if ((i%j)==0) {
            j++;
            long long d = __gcd(q,n);
            if (d == 1) continue;
            if (is_prime(d)) return d;
            if (d==4) return 2;
            if (d==9) return 3;
            return pollard_factor(d, (rand() & 32) - 16, rand() & 31);
        }
    }
    return -47;
}
```

```
// complete factorization wrapper: given n < 2^63, factor it
vector<ull> factor(ull n) {
    vector<ull> res;
    if (n<=1) return res;
    while (!is_prime(n)) {
        res.push_back( pollard_factor(n,-1,3) );
        n /= res.back();
    }
    res.push_back(n);
    sort(res.begin(),res.end());
    return res;
}
```

Vytúnené Eratostenovo sito:

```
const int SIEVE_MAX = 123456789;
#define ISPRIME(n) (!(__sito[(n)>>4] & (1<<(((n)>>1)&7)))) /* only works for odd values */
unsigned char __sito[ (SIEVE_MAX>>4) + 47 ];

void fill_sieve() {
    int __odm = int(3+sqrt(SIEVE_MAX));
    for (int i=3; i<=__odm; i+=2) if (ISPRIME(i)) {
        int j=i*i;
        while (j<SIEVE_MAX) {
            __sito[j>>4] |= 1 << ((j>>1)&7);
            j+=i<<1;
        }
    }
}

vector<int> get_primes() {
    vector<int> primes(1,2);
    for (int i=3; i<SIEVE_MAX; i+=2) if (ISPRIME(i)) primes.push_back(i);
    return primes;
}
```

12 Sufixové pole

Mega-stručná konštrukcia v $O(n \log^2 n)$ čase. Slušne rýchla (80/100 na SPOJ::SARRAY). Vo výstupnom poli `SA` sú indexy začiatkov sufixov usporiadané podľa abecedy. Teda napr. lexikograficky najmenší sufix začína v reťazci `S` na indexe `SA[0]`.
Konštrukcia longest common prefixov v $O(n)$ podľa článku *Kasai et al.: Linear-Time Longest-Common-Prefix Computation in Suffix Arrays and Its Applications*. Na políčku `lcp[i+1]` je uložená dĺžka spoločného prefixu pre sufixy ktoré v reťazci `S` začínajú na indexoch `SA[i]` a `SA[i+1]`. Otestované na SPOJ::TWICE.

```
vector<int> suffix_array(const string &S) {
    int sz=1, N=S.size();
    if (N == 0) return {};
    vector<int> SA(N), pos(N), tmp(N);
    for (int n=0; n<N; ++n) { SA[n] = n; pos[n] = S[n]; }

    auto cmp = [&](int i, int j) -> bool {
        if (pos[i] != pos[j]) return pos[i] < pos[j];
        i += sz; j += sz;
        return (i < N && j < N) ? pos[i] < pos[j] : i > j;
    };

    for (sz=1;;sz*=2) {
        sort( SA.begin(), SA.end(), cmp );
        for (int n=0; n<N-1; ++n) tmp[n+1] = tmp[n] + cmp( SA[n], SA[n+1] );
        for (int n=0; n<N; ++n) pos[ SA[n] ] = tmp[n];
        if (tmp[N-1] == N-1) break;
    }
    return SA;
}

vector<int> find_lcp(const string &S, const vector<int> &SA) {
    int N = S.size();
    vector<int> lcp(N), rank(N);
    for (int n=0; n<N; ++n) rank[ SA[n] ] = n;
    for (int i=0, h=0; i<N; ++i) if (rank[i] > 0) {
        int j = SA[ rank[i]-1 ];
        while (i+h<N && j+h<N && S[i+h]==S[j+h]) ++h;
        lcp[ rank[i] ] = h;
        if (h>0) --h;
    }
    return lcp;
}
```

13 Treap a multiset pomocou neho

V praktických testoch vyzeral tak dvakrát pomalšie ako klasický `multiset` (implementovaný cez red-black tree), ale navyše oproti `multiset`u má táto implementácia metódu `count(a,b)` čo spočíta prvky v intervale $[a, b)$ v log čase, a cez `[]` sa dá pristupovať k prvkom v usporiadanom poradí. Kvôli stručnejšej implementácii neuvoľňujeme pamäť, v prípade potreby treba doplniť `erase*`.

```
template<class T> struct TreapMultiset {
    struct item {
        T key;
        int priority, size;
        item *l, *r;
        item(const T &key) : key(key), priority(rand()), size(1), l(NULL), r(NULL) {}
    };

    typedef item* pitem;
    pitem root;

    // metody bezne volane zvonka

    int size() { return root ? root->size : 0; }

    void insert(T x) { insert(root,new item(x)); }

    void erase(T x) { erase(root,x); }

    void eraseall(T x) { pitem l,m,r; split(root,x,l,r); split(l,x,l,m,false); merge(root,l,r); }

    T operator[](int x) { return kth_smallest(root,x); }

    int count(T x) {
        pitem l,m,r;
        split(root,x,l,r); split(l,x,l,m,false);
        int answer = m ? m->size : 0;
        merge(l,l,m); merge(root,l,r);
        return answer;
    }

    int count_smaller(T x) {
        pitem l,r;
        split(root,x,l,r,false);
        int answer = l ? l->size : 0;
        merge(root,l,r);
        return answer;
    }

    int count_range(T x, T y) { return count_smaller(y)-count_smaller(x); } // half-open range [x,y)

    // interne metody

    TreapMultiset() : root(NULL) {}

    int get_size(pitem t) { return t ? t->size : 0; }

    void update(pitem t) { if (t) t->size = 1 + get_size(t->l) + get_size(t->r); }

    void split(pitem t, T key, pitem &l, pitem &r, bool equal_left = true) {
        if (!t) { l = r = NULL; return; }
        if (equal_left ? (key < t->key) : (key <= t->key)) {
            split(t->l, key, l, t->l, equal_left);
            r = t;
        } else {
            split(t->r, key, t->r, r, equal_left);
            l = t;
        }
        update(t);
    }

    void merge(pitem &t, pitem l, pitem r) {
        if (!l || !r) { t = l ? l : r; return; }
        if (l->priority > r->priority) {
            merge(l->r, l->r, r);
            t = l;
        } else {
            merge(r->l, l, r->l);
            t = r;
        }
        update(t);
    }

    void insert(pitem &t, pitem it) {
        if (!t) { t = it; return; }
        if (it->priority > t->priority) {
```

```
            split(t, it->key, it->l, it->r);
            t = it;
        } else {
            insert(it->key < t->key ? t->l : t->r, it);
        }
        update(t);
    }

    void erase(pitem &t, T key) {
        if (!t) return;
        if (t->key == key) merge(t, t->l, t->r);
        else erase(key < t->key ? t->l : t->r, key);
        update(t);
    }

    T kth_smallest(pitem p, int x) {
        int l = get_size(p->l);
        return l > x ? kth_smallest(p->l,x) : l == x ? p->key : kth_smallest(p->r,x-l-1);
    }

    void print(pitem t, int depth=0, ostream &out=cout) const {
        if (!t) return;
        out << string(2*depth,' ') << t->key << " (p " << t->priority << " s " << t->size << ")" << endl;
        out << string(2*depth,' ') << "left:" << endl;  print(t->l,depth+1);
        out << string(2*depth,' ') << "right:" << endl; print(t->r,depth+1);
    }
};

template<class T> ostream& operator<< (ostream &out, const TreapMultiset<T> &TM) { TM.print(TM.root,0,out); return out; }
```