Team Notebook

July 29, 2023

Contents

1 Strings **2**
- 1.1 AhoCorasick . . . 2
- 1.2 Manacher . . . 2
- 1.3 PalindromicTree . . . 2
- 1.4 PrefixFunction . . . 3
- 1.5 SuffixArray . . . 3
- 1.6 SuffixAutomaton . . . 5
- 1.7 hash . . . 6
- 1.8 hash2d . . . 6
- 1.9 sufarr . . . 7

2 dp **7**
- 2.1 convex-hull-trick . . . 7
- 2.2 divide-and-conquer . . . 8

3 geo2d **8**
- 3.1 circle . . . 8
- 3.2 convex-hull . . . 9
- 3.3 delaunay . . . 9
- 3.4 halfplane-intersect . . . 10
- 3.5 line . . . 10
- 3.6 point . . . 10
- 3.7 polygon . . . 11
- 3.8 sweep . . . 11
- 3.9 theorems . . . 12

4 graph **12**
- 4.1 bellman-ford . . . 12
- 4.2 dinic . . . 12
- 4.3 floyd-warshall . . . 12
- 4.4 heavy-light . . . 12
- 4.5 hungarian . . . 13
- 4.6 kuhn . . . 13
- 4.7 lca . . . 14
- 4.8 maxflow-mincost . . . 14
- 4.9 push-relabel . . . 14
- 4.10 strongly-connected-components . . . 15
- 4.11 two-sat . . . 15

5 implementation **15**
- 5.1 dsu . . . 15
- 5.2 mo . . . 16
- 5.3 persistent-segment-tree-lazy . . . 16
- 5.4 persistent-segment-tree . . . 16
- 5.5 segment-tree-lazy . . . 17
- 5.6 segment-tree . . . 18
- 5.7 sparse-table . . . 18
- 5.8 unordered-map . . . 18

6 imprimible **19**

7 math **19**
- 7.1 arithmetic . . . 19
- 7.2 crt . . . 19
- 7.3 discrete-log . . . 19
- 7.4 gauss . . . 19
- 7.5 matrix . . . 20
- 7.6 mod . . . 20
- 7.7 poly . . . 21
- 7.8 primes . . . 22
- 7.9 theorems . . . 23

1 Strings

1.1 AhoCorasick

```
#include<bits/stdc++.h>
using namespace std;

const int K = 26;
struct Vertex {
    int next[K];
    int leaf = 0;
    int leaf_id = -1;
    int p = -1;
    char pch;
    int link = -1;
    int exit = -1;
    int cnt = -1;
    int go[K];

    Vertex(int p=-1, char ch='$') : p(p), pch(ch) {
        fill(begin(next), end(next), -1);
        fill(begin(go), end(go), -1);
    }
};

vector<Vertex> t(1);

void add(string &s, int id) {
    int v = 0;
    for (char ch : s) {
        int c = ch - 'a';
        if (t[v].next[c] == -1) {
            t[v].next[c] = t.size();
            t.emplace_back(v, ch);
        }
        v = t[v].next[c];
    }
    t[v].leaf++;
    t[v].leaf_id = id;
}

int go(int v, char ch);

int get_link(int v) {
    if (t[v].link == -1) {
        if (v == 0 || t[v].p == 0)
            t[v].link = 0;
        else
            t[v].link = go(get_link(t[v].p), t[v].pch);
    }
    return t[v].link;
}

int go(int v, char ch) {
    int c = ch - 'a';
    if (t[v].go[c] == -1) {
        if (t[v].next[c] != -1)
            t[v].go[c] = t[v].next[c];
        else
            t[v].go[c] = v == 0 ? 0 : go(get_link(v), ch);
    }
    return t[v].go[c];
}

int next_match(int v)
{
    if(t[v].exit == -1)
    {
        if(t[get_link(v)].leaf)
            t[v].exit = get_link(v);
        else
            t[v].exit = v == 0 ? 0 : next_match(get_link(v));
    }
    return t[v].exit;
}

int cnt_matches(int v)
{
    if(t[v].cnt == -1)
        t[v].cnt = v == 0 ? 0 : t[v].leaf + cnt_matches(
            get_link(v));
    return t[v].cnt;
}
```

1.2 Manacher

```
#include<bits/stdc++.h>
using namespace std;
#define rep(i, n) for (int i = 0; i < (int)n; i++)
#define repx(i, a, b) for (int i = (int)a; i < (int)b; i++)

// odd[i] : length of the longest palindrome centered at i
// even[i] : length of the longest palindrome centered
    between i and i+1
void manacher(string &s, vector<int> &odd, vector<int> &even
    ) {
    string t = "$#";
    for(char c: s)
        t += c + string("#");
    t += "^";
    int n = t.size();
    vector<int> p(n);
    int l = 1, r = 1;
    repx(i, 1, n-1) {
        p[i] = max(0, min(r - i, p[l + (r - i)]));
        while(t[i - p[i]] == t[i + p[i]]) {
            p[i]++;
        }
        if(i + p[i] > r) {
            l = i - p[i], r = i + p[i];
        }
    }
    repx(i, 2, n-2) {
        if(i%2) even.push_back(p[i]-1);
        else odd.push_back(p[i]-1);
    }
}
```

1.3 PalindromicTree

```
#include<bits/stdc++.h>
using namespace std;
#define rep(i, n) for (int i = 0; i < (int)n; i++)
#define repx(i, a, b) for (int i = (int)a; i < (int)b; i++)

struct Node {
 int len;          // length of substring
 int edge[26];     // insertion edge for all characters a-z
 int link;         // the Maximum Palindromic Suffix Node
     for the current Node
 int i;            // (optional) start index of current Node
    int cnt = 1;      // (optional) number of occurrences of
        this substring
    Node(){ fill(begin(edge), end(edge), -1); }
};

struct EerTree {  // Palindromic Tree
    vector<Node> t; // tree
    int curr;     // current node

    EerTree(string &s) {
        t.resize(2);
        t.reserve(s.size()+2); // (optional) maximum size of
            tree
        t[0].len = -1;       // root 1
        t[0].link = 0;
        t[1].len = 0;        // root 2
        t[1].link = 0;
```

```
        curr = 1;
        rep(i, s.size()) insert(i, s); // construct tree

        // (optional) calculate number of occurrences of each
            node
        for(int i = t.size()-1; i > 1; i--)
            t[t[i].link].cnt += t[i].cnt;
    }

    void insert(int i, string &s) {
        int tmp = curr;
        while (i - t[tmp].len < 1 || s[i] != s[i-t[tmp].len
            -1])
            tmp = t[tmp].link;

        if(t[tmp].edge[s[i]-'a'] != -1){
            curr = t[tmp].edge[s[i]-'a'];  // node already
                exists
            t[curr].cnt++;                 // (optional)
                increase cnt
            return;
        }

        curr = t[tmp].edge[s[i]-'a'] = t.size(); // create
            new node
        t.emplace_back();

        t[curr].len = t[tmp].len + 2;      // set length
        t[curr].i = i - t[curr].len + 1;   // (optional) set
            start index

        if (t[curr].len == 1) {            // set suffix link
            t[curr].link = 1;
        } else {
            tmp = t[tmp].link;
            while (i-t[tmp].len < 1 || s[i] != s[i-t[tmp].len
                -1])
                tmp = t[tmp].link;
            t[curr].link = t[tmp].edge[s[i]-'a'];
        }
    }
};

int main()
{
 string s = "abcbab";
    EerTree pt(s);    // construct palindromic tree
 repx(i, 2, pt.t.size())  // list all distinct palindromes
 {
  cout << i-1 << ") ";
  repx(j, pt.t[i].i, pt.t[i].i + pt.t[i].len)
   cout << s[j];
  cout << " " << pt.t[i].cnt << endl;
 }

 return 0;
}
```

1.4 PrefixFunction

```
#include<bits/stdc++.h>
using namespace std;
#define rep(i, n) for (int i = 0; i < (int)n; i++)
#define repx(i, a, b) for (int i = (int)a; i < (int)b; i++)

vector<int> prefix_function(string s) {
    int n = s.size();
    vector<int> pi(n);
    repx(i, 1, n) {
        int j = pi[i-1];
        while (j > 0 && s[i] != s[j])
            j = pi[j-1];
        if (s[i] == s[j])
            j++;
        pi[i] = j;
    }
    return pi;
}

vector<vector<int>> aut;
void compute_automaton(string s) {
    s += '#';
    int n = s.size();
    vector<int> pi = prefix_function(s);
    aut.assign(n, vector<int>(26));
    rep(i, n) {
        rep(c, 26) {
            int j = i;
            while (j > 0 && 'a' + c != s[j])
                j = pi[j-1];
            if ('a' + c == s[j])
                j++;
            aut[i][c] = j;
        }
    }
}
```

1.5 SuffixArray

```
#include<bits/stdc++.h>
using namespace std;
#define LOG2(X) ((unsigned) (8*sizeof (unsigned long long) -
      __builtin_clzll((X)) - 1))
#define rep(i, n) for (int i = 0; i < (int)n; i++)
#define repx(i, a, b) for (int i = (int)a; i < (int)b; i++)

struct SuffixArray {
    int n; vector<int> C, R, R_, sa, sa_, lcp;
    inline int gr(int i) { return i < n ? R[i] : 0; } // sort
        suffixes
    //inline int gr(int i) { return R[i%n]; }    // sort
        cyclic shifts
    void csort(int maxv, int k) {
        C.assign(maxv + 1, 0); rep(i, n) C[gr(i + k)]++;
        repx(i, 1, maxv + 1) C[i] += C[i - 1];
        for (int i = n - 1; i >= 0; i--) sa_[--C[gr(sa[i] + k
            )]] = sa[i];
        sa.swap(sa_);
    }
    void getSA(vector<int>& s) {
        R = R_ = sa = sa_ = vector<int>(n); rep(i, n) sa[i] =
            i;
        sort(sa.begin(), sa.end(), [&s](int i, int j) {
            return s[i] < s[j]; });
        int r = R[sa[0]] = 1;
        repx(i, 1, n) R[sa[i]] = (s[sa[i]] != s[sa[i - 1]]) ?
            ++r : r;
        for (int h = 1; h < n && r < n; h <<= 1) {
            csort(r, h); csort(r, 0); r = R_[sa[0]] = 1;
            repx(i, 1, n) {
                if (R[sa[i]] != R[sa[i - 1]] || gr(sa[i] + h)
                    != gr(sa[i - 1] + h)) r++;
                R_[sa[i]] = r;
            } R.swap(R_);
        }
    }
    void getLCP(vector<int> &s) {// only works with suffixes
        (not cyclic shifts)
        lcp.assign(n, 0); int k = 0;
        rep(i, n) {
            int r = R[i] - 1;
            if (r == n - 1) { k = 0; continue; }
            int j = sa[r + 1];
            while (i + k < n && j + k < n && s[i + k] == s[j
                + k]) k++;
            lcp[r] = k; if (k) k--;
        }
```

```cpp
    }
    SuffixArray(vector<int> &s) { n = s.size(); getSA(s);
         getLCP(s); constructLCP(); }

    /* ------ Optional ------ */
    vector<vector<int>> T;
    void constructLCP() {
        T.assign(LOG2(n)+1, lcp);
        for(int k = 1; (1<<k) <= n; ++k)
            for(int i = 0; i + (1<<k) <= n; ++i)
                T[k][i] = min(T[k-1][i],T[k-1][i+(1<<(k-1))]);
    }
    // get LCP of suffix starting at i and suffix starting at
         j
    int queryLCP(int i, int j) {
        if(i == j) return n-i;
        i = R[i]-1; j = R[j]-1;
        if(i > j) swap(i, j);
        ll k = LOG2(j-i);
        return min(T[k][i],T[k][j-(1<<k)]);
    }
    // compare substring of length len1 starting at i
    // with substring of length len2 starting at j
    bool cmp(int i, int len1, int j, int len2) {
        if(queryLCP(i, j) >= min(len1, len2))
            return (len1 < len2);
        else
            return (R[i] < R[j]);
    }
};

vector<int> suffix_array;
vector<vector<int>> C;
int n;

void sort_cyclic_shifts(string s) {
    s += "$";
    n = s.size();
    const int alphabet = 256;
    vector<int> p(n), c(n), cnt(max(alphabet, n), 0);
    for (int i = 0; i < n; i++)
        cnt[s[i]]++;
    for (int i = 1; i < alphabet; i++)
        cnt[i] += cnt[i-1];
    for (int i = 0; i < n; i++)
        p[--cnt[s[i]]] = i;
    c[p[0]] = 0;
    int classes = 1;
    for (int i = 1; i < n; i++) {
        if (s[p[i]] != s[p[i-1]])
            classes++;
        c[p[i]] = classes - 1;
    }
    C.emplace_back(c.begin(), c.end());
    vector<int> pn(n), cn(n);
    for (int h = 0; (1 << h) < n; ++h) {
        for (int i = 0; i < n; i++) {
            pn[i] = p[i] - (1 << h);
            if (pn[i] < 0)
                pn[i] += n;
        }
        fill(cnt.begin(), cnt.begin() + classes, 0);
        for (int i = 0; i < n; i++)
            cnt[c[pn[i]]]++;
        for (int i = 1; i < classes; i++)
            cnt[i] += cnt[i-1];
        for (int i = n-1; i >= 0; i--)
            p[--cnt[c[pn[i]]]] = pn[i];
        cn[p[0]] = 0;
        classes = 1;
        for (int i = 1; i < n; i++) {
            pair<int, int> cur = {c[p[i]], c[(p[i] + (1 << h)
                 ) % n]};
            pair<int, int> prev = {c[p[i-1]], c[(p[i-1] + (1
                 << h)) % n]};
            if (cur != prev)
                ++classes;
            cn[p[i]] = classes - 1;
        }
        c.swap(cn);
        C.emplace_back(c.begin(), c.end());
    }
    p.erase(p.begin());
    suffix_array = p;
}

vector<int> lcp_construction(string &s, vector<int> &p) {
    int n = s.size();
    vector<int> rank(n);
    rep(i, n) rank[p[i]] = i;

    int k = 0;
    vector<int> lcp(n-1, 0);
    rep(i, n) {
        if (rank[i] == n - 1) {
            k = 0;
            continue;
        }
        int j = p[rank[i] + 1];
        while (i + k < n && j + k < n && s[i+k] == s[j+k])
            k++;
        lcp[rank[i]] = k;
        if (k)
            k--;
    }
    return lcp;
}

bool compare1(int i, int j, int l) {
    int k = LOG2(l);
    pair<int, int> a = {C[k][i], C[k][(i+l-(1 << k))%n]};
    pair<int, int> b = {C[k][j], C[k][(j+l-(1 << k))%n]};
    return a >= b;
}

bool compare2(int i, int j, int l) {
    int k = LOG2(l);
    pair<int, int> a = {C[k][i], C[k][(i+l-(1 << k))%n]};
    pair<int, int> b = {C[k][j], C[k][(j+l-(1 << k))%n]};
    return a <= b;
}


pair<int,int> find(int i, int len)
{
    int l = 0, r = suffix_array.size()-1;
    while(l != r)
    {
        int mid = (l+r)/2;
        if(compare1(suffix_array[mid], i, len))
            r = mid;
        else
            l = mid+1;
    }
    int left = l;

    l = 0, r = suffix_array.size()-1;
    while(l != r)
    {
        int mid = (l+r+1)/2;
        if(compare2(suffix_array[mid], i, len))
            l = mid;
        else
            r = mid-1;
    }
    int right = l;

    if(!compare1(suffix_array[left], i, len)) return {-1,-1};
    if(!compare2(suffix_array[right], i, len)) return
         {-1,-1};
```

```
    if(left > right) return {-1,-1};

    return {left, right};
}
```

1.6 SuffixAutomaton

```
#include<bits/stdc++.h>
using namespace std;
#define rep(i, n) for (int i = 0; i < (int)n; i++)
#define repx(i, a, b) for (int i = (int)a; i < (int)b; i++)

struct SuffixAutomaton {
    vector<map<char,int>> edges; // edges[i] : the labeled
        edges from node i
    vector<int> link;         // link[i]  : the suffix link
        of i
    vector<int> length;       // length[i] : the length of
        the longest string in the ith class
    vector<int> cnt;          // cnt[i]   : number of
        occurrences of each string in the ith class
    vector<int> paths;        // paths[i] : number of paths
        on the automaton starting from i
    vector<bool> terminal;    // terminal[i] : true if i is
        a terminal state
    vector<int> first_pos;
    vector<int> last_pos;
    int last;                 // the index of the
        equivalence class of the whole string

    SuffixAutomaton(string s) {
        edges.push_back(map<char,int>());
        link.push_back(-1);
        length.push_back(0);
        last = 0;

        rep(i, s.size()) { // construct r
            edges.push_back(map<char,int>());
            length.push_back(i+1);
            link.push_back(0);
            int r = edges.size() - 1;
            int p = last; // add edges to r and find p with
                link to q
            while(p >= 0 && !edges[p].count(s[i])) {
                edges[p][s[i]] = r;
                p = link[p];
            }
            if(p != -1) {
                int q = edges[p][s[i]];
                if(length[p] + 1 == length[q]) {
                    link[r] = q; // we do not have to split q,
                        just set the correct suffix link
                } else { // we have to split, add q'
                    edges.push_back(edges[q]); // copy edges
                        of q
                    length.push_back(length[p] + 1);
                    link.push_back(link[q]); // copy parent of
                        q
                    int qq = edges.size()-1;
                    link[q] = qq; // add qq as the new parent
                        of q and r
                    link[r] = qq;
                    while(p >= 0 && edges[p][s[i]] == q) { //
                        move short classes polling to q to
                        poll to q'
                        edges[p][s[i]] = qq;
                        p = link[p];
                    }
                }
            }
            last = r;
        }

/* ------ Optional ------ */

        // mark terminal nodes
        terminal.assign(edges.size(), false);
        int p = last;
        while(p > 0) {
            terminal[p] = true;
            p = link[p];
        }

        // precompute match count
        cnt.assign(edges.size(), -1);
        cnt_matches(0);

        // precompute number of paths (substrings) starting
            from state
        paths.assign(edges.size(), -1);
        cnt_paths(0);

        first_pos.assign(edges.size(), -1);
        get_first_pos(0);

        last_pos.assign(edges.size(), -1);
        get_last_pos(0);
}

int cnt_matches(int state) {
    if(cnt[state] != -1) return cnt[state];
    int ans = terminal[state];
    for(auto edge : edges[state])
        ans += cnt_matches(edge.second);
    return cnt[state] = ans;
}

int cnt_paths(int state) {
    if(paths[state] != -1) return paths[state];
    int ans = state == 0 ? 0 : 1; // without repetition (
        counts diferent substrings)
//  int ans = state == 0 ? 0 : cnt[state]; // with
    repetition
    for(auto edge : edges[state])
        ans += cnt_paths(edge.second);
    return paths[state] = ans;
}

int get_first_pos(int state) {
    if(first_pos[state] != -1) return first_pos[state];
    int ans = 0;
    for(auto edge : edges[state])
        ans = max(ans, get_first_pos(edge.second)+1);
    return first_pos[state] = ans;
}

int get_last_pos(int state) {
    if(last_pos[state] != -1) return last_pos[state];
    int ans = terminal[state] ? 0 : INT_MAX;//fix
    for(auto edge : edges[state])
        ans = min(ans, get_last_pos(edge.second)+1);
    return last_pos[state] = ans;
}

string get_k_substring(int k) // 0-indexed
{
    string ans;
    int state = 0;
    while(true)
    {
        int curr = state == 0 ? 0 : 1; // without
            repetition (counts diferent substrings)
    //  int curr = state == 0 ? 0 : cnt[state]; // with
        repetition
        if(curr > k) return ans;
        k -= curr;

        for(auto edge : edges[state]) {
            if(paths[edge.second] <= k) {
```

```
                k -= paths[edge.second];
            } else {
                ans += edge.first;
                state = edge.second;
                break;
            }
        }
    }
  }
};
```

1.7 hash

```
// compute substring hashes in O(1).
// hashes are compatible between different strings.
struct Hash {
    ll HMOD;
    int N;
    vector<int> h;
    vector<int> p;

    Hash() {}
    // O(N)
    Hash(const string &s, ll HMOD_ = 1000003931)
        : N(s.size() + 1), HMOD(HMOD_), p(N), h(N) {
        static const ll P =
            chrono::steady_clock::now().time_since_epoch().
                count() % (1 << 29);
        p[0] = 1;
        rep(i, N - 1) p[i + 1] = p[i] * P % HMOD;
        rep(i, N - 1) h[i + 1] = (h[i] + (ll)s[i] * p[i]) %
             HMOD;
    }

    // O(1)
    pair<ll, int> get(int i, int j) { return {(h[j] - h[i] +
         HMOD) % HMOD, i}; }

    bool cmp(pair<ll, int> x0, pair<ll, int> x1) {
        int d = x0.second - x1.second;
        ll &lo = d < 0 ? x0.first : x1.first;
        lo = lo * p[abs(d)] % HMOD;
        return x0.first == x1.first;
    }
};

// compute hashes in multiple prime modulos simultaneously,
     to reduce the chance
// of collisions.
struct HashM {
    int N;
    vector<Hash> sub;

    HashM() {}
    // O(K N)
    HashM(const string &s, const vector<ll> &mods) : N(mods.
         size()), sub(N) {
        rep(i, N) sub[i] = Hash(s, mods[i]);
    }

    // O(K)
    vector<pair<ll, int>> get(int i, int j) {
        vector<pair<ll, int>> hs(N);
        rep(k, N) hs[k] = sub[k].get(i, j);
        return hs;
    }

    bool cmp(const vector<pair<ll, int>> &x0, const vector<
         pair<ll, int>> &x1) {
        rep(i, N) if (!sub[i].cmp(x0[i], x1[i])) return false
            ;
        return true;
    }

    bool cmp(int i0, int j0, int i1, int j1) {
        rep(i, N) if (!sub[i].cmp(sub[i].get(i0, j0),
                                  sub[i].get(i1, j1))) return
                                       false;
        return true;
    }
};

#ifndef NOMAIN_HASH

int main() {
    const vector<ll> HMOD = {1000001237, 1000003931};
    //  0123456789012345678901 2
    string s = "abracadabra abracadabra";
    HashM h(s, HMOD);
    rep(i0, s.size() + 1) repx(j0, i0, s.size() + 1) rep(i1,
         s.size() + 1)
        repx(j1, i1, s.size() + 1) {
        bool eq = h.cmp(h.get(i0, j0), h.get(i1, j1));
        bool eq2 = s.substr(i0, j0 - i0) == s.substr(i1, j1 -
             i1);
        if (eq != eq2) {
            cout << " hash says strings \"" << s.substr(i0,
                 j0 - i0) << "\" "
                 << (eq ? "==" : "!=") << " \"" << s.substr(i1
                     , j1 - i1)
                 << "\" but in reality they are " << (eq2 ? "
                     ==" : "!=")
                 << endl;
        }
    }
}

#endif
```

1.8 hash2d

```
using Hash = pair<ll, int>;

struct Block {
    int x0, y0, x1, y1;
};

struct Hash2d {
    ll HMOD;
    int W, H;
    vector<int> h;
    vector<int> p;

    Hash2d() {}
    Hash2d(const string &s, int W_, int H_, ll HMOD_ =
         1000003931)
        : W(W_ + 1), H(H_ + 1), HMOD(HMOD_) {
        static const ll P =
            chrono::steady_clock::now().time_since_epoch().
                count() % (1 << 29);
        p.resize(W * H);
        p[0] = 1;
        rep(i, W * H - 1) p[i + 1] = p[i] * P % HMOD;
        h.assign(W * H, 0);
        repx(y, 1, H) repx(x, 1, W) {
            ll c = (ll)s[(y - 1) * (W - 1) + x - 1] * p[y * W
                 + x] % HMOD;
            h[y * W + x] = (HMOD + h[y * W + x - 1] + h[(y -
                1) * W + x] -
                          h[(y - 1) * W + x - 1] + c) %
                         HMOD;
        }
    }

    bool isout(Block s) {
        return s.x0 < 0 || s.x0 >= W || s.x1 < 0 || s.x1 >= W
             || s.y0 < 0 ||
```

```
            s.y0 >= H || s.y1 < 0 || s.y1 >= H;
    }

    Hash get(Block s) {
        return {(2 * HMOD + h[s.y1 * W + s.x1] - h[s.y1 * W +
              s.x0] -
                h[s.y0 * W + s.x1] + h[s.y0 * W + s.x0]) %
                  HMOD,
               s.y0 * W + s.x0};
    }

    bool cmp(Hash x0, Hash x1) {
        int d = x0.second - x1.second;
        ll &lo = d < 0 ? x0.first : x1.first;
        lo = lo * p[abs(d)] % HMOD;
        return x0.first == x1.first;
    }
};

struct Hash2dM {
    int N;
    vector<Hash2d> sub;

    Hash2dM() {}
    Hash2dM(const string &s, int W, int H, const vector<ll> &
         mods)
        : N(mods.size()), sub(N) {
        rep(i, N) sub[i] = Hash2d(s, W, H, mods[i]);
    }

    bool isout(Block s) { return sub[0].isout(s); }

    vector<Hash> get(Block s) {
        vector<Hash> hs(N);
        rep(i, N) hs[i] = sub[i].get(s);
        return hs;
    }

    bool cmp(const vector<Hash> &x0, const vector<Hash> &x1)
         {
        rep(i, N) if (!sub[i].cmp(x0[i], x1[i])) return false
             ;
        return true;
    }

    bool cmp(Block s0, Block s1) {
        rep(i, N) if (!sub[i].cmp(sub[i].get(s0), sub[i].get(
             s1))) return false;
        return true;
    }
};

#ifndef NOMAIN_HASH2D

const vector<ll> HMOD = {1000002649, 1000000933, 1000003787,
     1000002173};

int main() {}

#endif
```

1.9 sufarr

```
// build the suffix array
// suffixes are sorted, with each suffix represented by its
     starting position
vector<int> suffixarray(const string &s) {
    int N = s.size() + 1; // optional: include terminating
         NUL
    vector<int> p(N), p2(N), c(N), c2(N), cnt(256);
    rep(i, N) cnt[s[i]] += 1;
    repx(b, 1, 256) cnt[b] += cnt[b - 1];
    rep(i, N) p[--cnt[s[i]]] = i;
    repx(i, 1, N) c[p[i]] = c[p[i - 1]] + (s[p[i]] != s[p[i -
         1]]);
    for (int k = 1; k < N; k <<= 1) {
        int C = c[p[N - 1]] + 1;
        cnt.assign(C + 1, 0);
        for (int &pi : p) pi = (pi - k + N) % N;
        for (int cl : c) cnt[cl + 1] += 1;
        rep(i, C) cnt[i + 1] += cnt[i];
        rep(i, N) p2[cnt[c[p[i]]]++] = p[i];
        c2[p2[0]] = 0;
        repx(i, 1, N) c2[p2[i]] =
            c2[p2[i - 1]] + (c[p2[i]] != c[p2[i - 1]] ||
                             c[(p2[i] + k) % N] != c[(p2[i - 1]
                                 + k) % N]);
        swap(c, c2), swap(p, p2);
    }
    p.erase(p.begin()); // optional: erase terminating NUL
    return p;
}

// build the lcp
// `lcp[i]` represents the length of the longest common
     prefix between suffix i
// and suffix i+1 in the suffix array `p`. the last element
     of `lcp` is zero by
// convention
vector<int> makelcp(const string &s, const vector<int> &p) {
    int N = p.size(), k = 0;
    vector<int> r(N), lcp(N);
    rep(i, N) r[p[i]] = i;
    rep(i, N) {
        if (r[i] + 1 >= N) {
            k = 0;
            continue;
        }
        int j = p[r[i] + 1];
        while (i + k < N && j + k < N && s[i + k] == s[j + k
             ]) k += 1;
        lcp[r[i]] = k;
        if (k) k -= 1;
    }
    return lcp;
}

#ifndef NOMAIN_SUFARR

void test(const string &s) {
    cout << "suffix array for string \"" << s << "\" (length
         " << s.size()
         << "):" << endl;
    vector<int> sa = suffixarray(s);
    vector<int> lcp = makelcp(s, sa);
    rep(i, sa.size()) {
        int j = sa[i];
        if (i > 0) cout << "  " << lcp[i - 1] << endl;
        cout << " \"" << s.substr(j) << "\"" << endl;
    }
}

int main() {
    test("hello");
    test("abracadabra");
}

#endif
```

2 dp

2.1 convex-hull-trick

```
struct Line {
    mutable ll a, b, c;

    bool operator<(Line r) const { return a < r.a; }
```

```
    bool operator<(ll x) const { return c < x; }
};

// dynamically insert ‘a*x + b‘ lines and query for maximum
     at any x
// all operations have complexity O(log N)
struct LineContainer : multiset<Line, less<>> {

    ll div(ll a, ll b) {
        return a / b - ((a ^ b) < 0 && a % b);
    }

    bool isect(iterator x, iterator y) {
        if (y == end()) return x->c = INF, 0;
        if (x->a == y->a) x->c = x->b > y->b ? INF : -INF;
        else x->c = div(y->b - x->b, x->a - y->a);
        return x->c >= y->c;
    }

    void add(ll a, ll b) {
        // a *= -1, b *= -1 // for min
        auto z = insert({a, b, 0}), y = z++, x = y;
        while (isect(y, z)) z = erase(z);
        if (x != begin() && isect(--x, y)) isect(x, y = erase
             (y));
        while ((y = x) != begin() && (--x)->c >= y->c) isect(
             x, erase(y));
    }

    ll query(ll x) {
        if (empty()) return -INF; // INF for min
        auto l = *lower_bound(x);
        return l.a * x + l.b;
        // return -l.a * x - l.b; // for min
    }
};
```

2.2 divide-and-conquer

```
// for every index i assign an optimal index j, such that
     cost(i, j) is
// minimal for every i. the property that if i2 >= i1 then
     j2 >= j1 is
// exploited (monotonic condition).
// calculate optimal index for all indices in range [l, r)
     knowing that
// the optimal index for every index in this range is within
      [optl, optr).
// time: O(N log N)
void calc(vector<int> &opt, int l, int r, int optl, int optr
     ) {
    if (l == r) return;
    int i = (l + r) / 2;
    ll optc = INF;
    int optj;
    repx(j, optl, optr) {
        ll c = i + j; // cost(i, j)
        if (c < optc) optc = c, optj = j;
    }
    opt[i] = optj;
    calc(opt, l, i, optl, optj + 1);
    calc(opt, i + 1, r, optj, optr);
}
```

3 geo2d

3.1 circle

```
struct C {
    P o; T r;

    C(P o, T r) : o(o), r(r) {}
    C() : C(P(), T()) {}

    // intersects the circle with a line, assuming they
         intersect
    // results are sorted with respect to the direction of
         the line
    pair<P, P> line_inter(L l) const {
        P c = l.closest_to(o);
        T c2 = (c - o).magsq();
        P e = sqrt(max(r * r - c2, T())) * l.d.unit();
        return {c - e, c + e};
    }

    // checks whether the given line collides with the circle
    // negative: 2 intersections
    // zero: 1 intersection
    // positive: 0 intersections
    T line_collide(L l) const {
        T c2 = (l.closest_to(o) - o).magsq();
        return c2 - r * r;
    }

    // calculates the two intersections between two circles
    // the circles must intersect in one or two points!
    pair<P, P> inter(C h) const {
        P d = h.o - o;
        T c = (r * r - h.r * h.r) / d.magsq();
        return h.line_inter({(1 + c) / 2 * d, d.rot()});
    }

    // check if the given circles intersect
    bool collide(C h) const {
        return (h.o - o).magsq() <= (h.r + r) * (h.r + r);
    }

    // get one of the two tangents that cross through the
         point
    // the point must not be inside the circle
    // a = -1: cw (relative to the circle) tangent
    // a = 1: ccw (relative to the circle) tangent
    P point_tangent(P p, T a) const {
        T c = r * r / p.magsq();
        return o + c * (p - o) - a * sqrt(c * (1 - c)) * (p -
              o).rot();
    }

    // get one of the 4 tangents between the two circles
    // a = 1: exterior tangents
    // a = -1: interior tangents (requires no area overlap)
    // b = 1: ccw tangent
    // b = -1: cw tangent
    // the line origin is on this circumference, and the
         direction
    // is a unit vector towards the other circle
    L tangent(C c, T a, T b) const {
        T dr = a * r - c.r;
        P d = c.o - o;
        P n = (d * dr + b * d.rot() * sqrt(d.magsq() - dr *
             dr)).unit();
        return {o + n * r, -b * n.rot()};
    }

    // find the circumcircle of the given **non-degenerate**
         triangle
    static C thru_points(P a, P b, P c) {
        L l((a + b) / 2, (b - a).rot());
        P p = l.intersection(L((a + c) / 2, (c - a).rot()));
        return {p, (p - a).mag()};
    }

    // find the two circles that go through the given point,
         are tangent
    // to the given line and have radius ‘r‘
    // the point-line distance must be at most ‘r‘!
    // the circles are sorted in the direction of the line
```

```
    static pair<C, C> thru_point_line_r(P a, L t, T r) {
        P d = t.d.rot().unit();
        if (d * (a - t.o) < 0) d = -d;
        auto p = C(a, r).line_inter({t.o + d * r, t.d});
        return {{p.first, r}, {p.second, r}};
    }

    // find the two circles that go through the given points
        and have
    // radius 'r'
    // the circles are sorted by angle with respect to the
        first point
    // the points must be at most at distance 'r'!
    static pair<C, C> thru_points_r(P a, P b, T r) {
        auto p = C(a, r).line_inter({(a + b) / 2, (b - a).rot
            ()});
        return {{p.first, r}, {p.second, r}};
    }
};
```

3.2 convex-hull

```
// get the convex hull with the least amount of vertices for
    the given set
// of points
// probably misbehaves if points are not all distinct!
vector<P> convex_hull(vector<P> &ps) {
    int N = ps.size(), n = 0, k = 0;
    if (N <= 2) return ps;
    rep(i, N) if (make_pair(ps[i].y, ps[i].x) < make_pair(ps[
        k].y, ps[k].x)) k = i;
    swap(ps[k], ps[0]);
    sort(++ps.begin(), ps.end(), [&](P l, P r) {
        T x = (r - l) / (ps[0] - l), d = (r - l) * (ps[0] - l
            );
        return x > 0 || x == 0 && d < 0;
    });

    vector<P> H;
    for (P p : ps) {
        while (n >= 2 && (H[n - 1] - p) / (H[n - 2] - p) >=
            0) H.pop_back(), n--;
        H.push_back(p), n++;
    }
    return H;
}
```

3.3 delaunay

```
typedef __int128_t lll; // if on a 64-bit platform

struct Q {
    Q *rot, *o; P p = {INF, INF}; bool mark;
    P &F() { return r()->p; }
    Q *&r() { return rot->rot; }
    Q *prev() { return rot->o->rot; }
    Q *next() { return r()->prev(); }
};

T cross(P a, P b, P c) { return (b - a) / (c - a); }

bool circ(P p, P a, P b, P c) { // is p in the circumcircle?
    lll p2 = p.magsq(), A = a.magsq() - p2,
        B = b.magsq() - p2, C = c.magsq() - p2;
    return cross(p, a, b) * C + cross(p, b, c) * A + cross(p,
        c, a) * B > 0;
}

Q *makeEdge(Q *&H, P orig, P dest) {
    Q *r = H ? H : new Q{new Q{new Q{new Q{0}}}};
    H = r->o; r->r()->r() = r;
    repx(i, 0, 4) r = r->rot, r->p = {INF, INF},
        r->o = i & 1 ? r : r->r();
    r->p = orig; r->F() = dest;
    return r;
}

void splice(Q *a, Q *b) {
    swap(a->o->rot->o, b->o->rot->o); swap(a->o, b->o);
}

Q *connect(Q *&H, Q *a, Q *b) {
    Q *q = makeEdge(H, a->F(), b->p);
    splice(q, a->next()); splice(q->r(), b); return q;
}

pair<Q *, Q *> rec(Q *&H, const vector<P> &s) {
    if (s.size() <= 3) {
        Q *a = makeEdge(H, s[0], s[1]), *b = makeEdge(H, s
            [1], s.back());
        if (s.size() == 2) return {a, a->r()}; splice(a->r(),
             b);
        auto side = cross(s[0], s[1], s[2]);
        Q *c = side ? connect(H, b, a) : 0;
        return {side < 0 ? c->r() : a, side < 0 ? c : b->r()
            };
    }
#define J(e) e->F(), e->p
#define valid(e) (cross(e->F(), J(base)) > 0)
    Q *A, *B, *ra, *rb; int half = s.size() / 2;
    tie(ra, A) = rec(H, {s.begin(), s.end() - half});
    tie(B, rb) = rec(H, {s.begin() + s.size() - half, s.end()
        });
    while ((cross(B->p, J(A)) < 0 && (A = A->next())) ||
           (cross(A->p, J(B)) > 0 && (B = B->r()->o)));
    Q *base = connect(H, B->r(), A);
    if (A->p == ra->p) ra = base->r();
    if (B->p == rb->p) rb = base;

#define DEL(e, init, dir) Q *e = init->dir; \
    if (valid(e)) while (circ(e->dir->F(), J(base), e->F()))
        { \
            Q *t = e->dir; splice(e, e->prev()); \
            splice(e->r(), e->r()->prev()); e->o = H; H = e;
                e = t; \
        }
    for (;;) {
        DEL(LC, base->r(), o); DEL(RC, base, prev());
        if (!valid(LC) && !valid(RC)) break;
        if (!valid(LC) || (valid(RC) && circ(J(RC), J(LC))))
            base = connect(H, RC, base->r());
        else base = connect(H, base->r(), LC->r());
    }
    return {ra, rb};
#undef J
#undef valid
#undef DEL
}

// there must be no duplicate points
// returns no triangles in the case of all collinear points
// produces counter-clockwise triangles ordered in triples
// maximizes the minimum angle across all triangulations
// the euclidean mst is a subset of these edges
// O(N log N)
vector<P> triangulate(vector<P> pts) {
    sort(pts.begin(), pts.end(), [](P a, P b) {
        return make_pair(a.x, a.y) < make_pair(b.x, b.y);
    });
    assert(unique(pts.begin(), pts.end()) == pts.end());
    if (pts.size() < 2) return {};
    Q *H = 0; Q *e = rec(H, pts).first;
    vector<Q *> q = {e}; int qi = 0;
    while (cross(e->o->F(), e->F(), e->p) < 0) e = e->o;
#define ADD                             \
    {                                   \
```

```
        Q *c = e;               \
        do {                    \
            c->mark = 1; pts.push_back(c->p); \
            q.push_back(c->r()); c = c->next(); \
        } while (c != e);       \
    }
    ADD;
    pts.clear();
    while (qi < (int)q.size()) if (!(e = q[qi++])->mark) ADD;
    return pts;
#undef ADD
}
```

3.4 halfplane-intersect

```
// obtain the convex polygon that results from intersecting
    the given list
// of halfplanes, represented as lines that allow their left
    side
// assumes the halfplane intersection is bounded
vector<P> halfplane_intersect(vector<L> &H) {
    L bb(P(-INF, -INF), P(INF, 0));
    rep(k, 4) H.push_back(bb), bb.o = bb.o.rot(), bb.d = bb.d
        .rot();

    sort(begin(H), end(H), [](L a, L b) { return a.d.angcmp(b
        .d) < 0; });
    deque<L> q; int n = 0;
    rep(i, H.size()) {
        while (n >= 2 && H[i].side(q[n - 1].intersection(q[n
            - 2])) > 0)
            q.pop_back(), n--;
        while (n >= 2 && H[i].side(q[0].intersection(q[1])) >
            0)
            q.pop_front(), n--;
        if (n > 0 && H[i].parallel(q[n - 1])) {
            if (H[i].d * q[n - 1].d < 0) return {};
            if (H[i].side(q[n - 1].o) > 0) q.pop_back(), n--;
            else continue;
        }
        q.push_back(H[i]), n++;
    }

    while (n >= 3 && q[0].side(q[n - 1].intersection(q[n -
        2])) > 0)
        q.pop_back(), n--;
    while (n >= 3 && q[n - 1].side(q[0].intersection(q[1])) >
        0)
        q.pop_front(), n--;
    if (n < 3) return {};

    vector<P> ps(n);
    rep(i, n) ps[i] = q[i].intersection(q[(i + 1) % n]);
    return ps;
}
```

3.5 line

```
// a segment or an infinite line
// does not handle point segments correctly!
struct L {
    P o, d;
    L() : o(), d() {}
    L(P o, P d) : o(o), d(d) {}

    L(P ab, T c) : d(ab.rot()), o(ab * -c / ab.magsq()) {}
    pair<P, T> line_eq() { return {-d.rot(), d.rot() * o}; }

    // returns a number indicating which side of the line the
        point is in
    // negative: left, positive: right
    T side(P r) const { return (r - o) / d; }

    // returns the intersection coefficient
    // in the range [0, d / r.d]
    // if d / r.d is zero, the lines are parallel
    T inter(L r) const { return (r.o - o) / r.d; }

    // get the single intersection point
    // lines must not be parallel
    P intersection(L r) const { return o + d * inter(r) / (d
        / r.d); }

    // check if lines are parallel
    bool parallel(L r) const { return abs(d / r.d) <= EPS; }

    // check if segments intersect
    bool seg_collide(L r) const {
        T z = d / r.d;
        if (abs(z) <= EPS) {
            if (abs(side(r.o)) > EPS) return false;
            T s = (r.o - o) * d, e = s + r.d * d;
            if (s > e) swap(s, e);
            return s <= d * d + EPS && e >= -EPS;
        }
        T s = inter(r), t = -r.inter(*this);
        if (z < 0) s = -s, t = -t, z = -z;
        return s >= -EPS && s <= z + EPS && t >= -EPS && t <=
            z + EPS;
    }

    // full segment intersection
    // produces a point segment if the intersection is a
        point
    // however it **does not** handle point segments as input
        !
    bool seg_inter(L r, L *out) const {
        T z = d / r.d;
        if (abs(z) <= EPS) {
            if (abs(side(r.o)) > EPS) return false;
            if (r.d * d < 0) r = {r.o + r.d, -r.d};
            P s = o * d < r.o * d ? r.o : o;
            P e = (o + d) * d < (r.o + r.d) * d ? o + d : r.o
                + r.d;
            if (s * d > e * d) return false;
            return *out = L(s, e - s), true;
        }
        T s = inter(r), t = -r.inter(*this);
        if (z < 0) s = -s, t = -t, z = -z;
        if (s >= -EPS && s <= z + EPS && t >= -EPS && t <= z
            + EPS)
            return *out = L(o + d * s / z, P()), true;
        return false;
    }

    // check if the given point is on the segment
    bool point_on_seg(P r) const {
        if (abs(side(r)) > EPS) return false;
        if ((r - o) * d < -EPS) return false;
        if ((r - o - d) * d > EPS) return false;
        return true;
    }

    // get the point in this line that is closest to a given
        point
    P closest_to(P r) const {
        P dr = d.rot(); return r + (o - r) * dr * dr / d.
            magsq();
    }
};
```

3.6 point

```
struct P {
    T x, y;
    P(T x, T y) : x(x), y(y) {}
```

```
    P() : P(0, 0) {}

    friend ostream &operator<<(ostream &s, const P &r) {
        return s << r.x << " " << r.y;
    }
    friend istream &operator>>(istream &s, P &r) { return s
        >> r.x >> r.y; }

    P operator+(P r) const { return {x + r.x, y + r.y}; }
    P operator-(P r) const { return {x - r.x, y - r.y}; }
    P operator*(T r) const { return {x * r, y * r}; }
    P operator/(T r) const { return {x / r, y / r}; }
    P operator-() const { return {-x, -y}; }
    friend P operator*(T l, P r) { return {l * r.x, l * r.y};
        }

    P rot() const { return {-y, x}; }
    T operator*(P r) const { return x * r.x + y * r.y; }
    T operator/(P r) const { return rot() * r; }

    T magsq() const { return x * x + y * y; }
    T mag() const { return sqrt(magsq()); }
    P unit() const { return *this / mag(); }

    bool half() const { return abs(y) <= EPS && x < -EPS || y
        < -EPS; }
    T angcmp(P r) const {
        int h = (int)half() - r.half();
        return h ? h : r / *this;
    }

    bool operator==(P r) const { return abs(x - r.x) <= EPS
        && abs(y - r.y) <= EPS; }

    double angle() const { return atan2(y, x); }
    static P from_angle(double a) { return {cos(a), sin(a)};
        }
};
```

3.7 polygon

```
// get the area of a simple polygon in ccw order
T area(const vector<P> &ps) {
    int N = ps.size();
    T a = 0;
    rep(i, N) a += (ps[i] - ps[0]) / (ps[(i + 1) % N] - ps[i
        ]);
    return a / 2;
}

// checks whether a point is inside a simple polygon
// returns -1 if inside, 0 if on border, 1 if outside
// O(N)
int in_poly(const vector<P> &ps, P p) {
    int N = ps.size(), w = 0;
    rep(i, N) {
        P s = ps[i] - p, e = ps[(i + 1) % N] - p;
        if (s == P()) return 0;
        if (s.y == 0 && e.y == 0) {
            if (min(s.x, e.x) <= 0 && 0 <= max(s.x, e.x))
                return 0;
        } else {
            bool b = s.y < 0;
            if (b != (e.y < 0)) {
                T z = s / e; if (z == 0) return 0;
                if (b == (z > 0)) w += b ? 1 : -1;
            }
        }
    }
    return w ? -1 : 1;
}

// check if a point is in a convex polygon
struct InConvex {
    vector<P> ps;
    T ll, lh, rl, rh;
    int N, m;

    // preprocess polygon
    // O(N)
    InConvex(const vector<P> &p) : ps(p), N(ps.size()), m(0)
        {
        assert(N >= 2);
        rep(i, N) if (ps[i].x < ps[m].x) m = i;
        rotate(ps.begin(), ps.begin() + m, ps.end());
        rep(i, N) if (ps[i].x > ps[m].x) m = i;
        ll = lh = ps[0].y, rl = rh = ps[m].y;
        for (P p : ps) {
            if (p.x == ps[0].x) ll = min(ll, p.y), lh = max(
                lh, p.y);
            if (p.x == ps[m].x) rl = min(rl, p.y), rh = max(
                rh, p.y);
        }
    }
    InConvex() {}

    // check if point belongs in polygon
    // returns -1 if inside, 0 if on border, 1 if outside
    // O(log N)
    int in_poly(P p) {
        if (p.x < ps[0].x || p.x > ps[m].x) return 1;
        if (p.x == ps[0].x) return p.y < ll || p.y > lh;
        if (p.x == ps[m].x) return p.y < rl || p.y > rh;
        int r = upper_bound(ps.begin(), ps.begin() + m, p,
            [](P a, P b) { return a.x < b.x; }) - ps.begin();
        T z = (ps[r - 1] - ps[r]) / (p - ps[r]); if (z >= 0)
            return !!z;
        r = upper_bound(ps.begin() + m, ps.end(), p,
            [](P a, P b) { return a.x > b.x; }) - ps.begin();
        z = (ps[r - 1] - ps[r % N]) / (p - ps[r % N]);
        if (z >= 0) return !!z; return -1;
    }
};
```

3.8 sweep

```
#include "point.cpp"

// iterate over all pairs of points
// `op` is called with all ordered pairs of different
    indices `(i, j)`
// additionally, the `ps` vector is kept sorted by signed
    distance
// to the line formed by `i` and `j`
// for example, if the vector from `i` to `j` is pointing
    right,
// the `ps` vector is sorted from smallest `y` to largest `y
    `
// note that, because the `ps` vector is sorted by signed
    distance,
// `j` is always equal to `i + 1`
// this means that the amount of points to the left of the
    line is always `N - i`
template <class OP>
void all_pair_points(vector<P> &ps, OP op) {
    int N = ps.size();
    sort(ps.begin(), ps.end(), [](P a, P b) {
        return make_pair(a.y, a.x) < make_pair(b.y, b.x);
    });
    vector<pair<int, int>> ss;
    rep(i, N) rep(j, N) if (i != j) ss.push_back({i, j});
    stable_sort(ss.begin(), ss.end(), [&](auto a, auto b) {
        return (ps[a.second] - ps[a.first]).angle_lt(ps[b.
            second] - ps[b.first]);
    });
    vector<int> p(N); rep(i, N) p[i] = i;
    for (auto [i, j] : ss)
```

```
    { op(p[i], p[j]); swap(ps[p[i]], ps[p[j]]); swap(p[i
        ], p[j]); }
}
```

3.9 theorems

```
// Pick's theorem
//     Simple polygon with integer vertices:
//     A = I + B / 2 - 1
//     A: Area of the polygon
//     I: Integer points strictly inside the polygon
//     B: Integer points on the boundary of the polygon
```

4 graph

4.1 bellman-ford

```
struct Edge { int u, v; ll w; };

// find distance from source node to all nodes.
// supports negative edge weights.
// returns true if a negative cycle is detected.
//
// time: O(V E)
bool bellman_ford(int N, int s, vector<Edge> &E, vector<ll>
     &D, vector<int> &P) {
    P.assign(N, -1), D.assign(N, INF), D[s] = 0;
    rep(i, N - 1) {
        bool f = true;
        rep(ei, E.size()) {
            auto &e = E[ei];
            ll n = D[e.u] + e.w;
            if (D[e.u] < INF && n < D[e.v])
                D[e.v] = n, P[e.v] = ei, f = false;
        }
        if (f) return false;
    }
    return true;
}
```

4.2 dinic

```
struct Edge { int u, v; ll c, f = 0; };

// maximum flow algorithm.
// time: O(E V^2)
//      O(E V^(2/3)) / O(E sqrt(E)) unit capacities
//      O(E sqrt(V))                unit networks (hopcroft-
     karp)
// unit network: c in {0, 1} and forall v, len(incoming(v))
     <= 1 or len(outgoing(v)) <= 1
// min-cut: find all nodes reachable from the source in the
     residual graph
struct Dinic {
    int N, s, t; vector<vector<int>> G;
    vector<Edge> E; vector<int> lvl, ptr;
    Dinic() {}
    Dinic(int N, int s, int t) : N(N), s(s), t(t), G(N) {}

    void add_edge(int u, int v, ll c) {
        G[u].push_back(E.size()); E.push_back({u, v, c});
        G[v].push_back(E.size()); E.push_back({v, u, 0});
    }

    ll push(int u, ll p) {
        if (u == t || p <= 0) return p;
        while (ptr[u] < G[u].size()) {
            int ei = G[u][ptr[u]++];
            Edge &e = E[ei];
            if (lvl[e.v] != lvl[u] + 1) continue;
            ll a = push(e.v, min(e.c - e.f, p));
            if (a <= 0) continue; e.f += a, E[ei ^ 1].f -= a;
            return a;
        }
        return 0;
    }

    ll maxflow() {
        ll f = 0;
        while (true) {
            // bfs to build levels
            lvl.assign(N, -1); queue<int> q; lvl[s] = 0, q.
                push(s);
            while (!q.empty()) {
                int u = q.front(); q.pop();
                for (int ei : G[u]) {
                    Edge &e = E[ei];
                    if (e.c - e.f <= 0 || lvl[e.v] != -1)
                        continue;
                    lvl[e.v] = lvl[u] + 1, q.push(e.v);
                }
            }
            if (lvl[t] == -1) break;

            // dfs to find blocking flow
            ptr.assign(N, 0); while (ll ff = push(s, INF)) f
                += ff;
        }
        return f;
    }
};
```

4.3 floyd-warshall

```
// O(V^3) time and O(V^2) memory.
// requires an NxN array to store results.
// works with negative edges, but not negative cycles.
void floyd(const vector<vector<pair<int, ll>>> &G, vector<
     vector<ll>> &dists) {
    int N = G.size();
    rep(i, N) rep(j, N) dists[i][j] = i == j ? 0 : INF;
    rep(i, N) for (auto edge : G[i]) dists[i][edge.first] =
         edge.second;
    rep(k, N) rep(i, N) rep(j, N)
        dists[i][j] = min(dists[i][j], dists[i][k] + dists[k
            ][j]);
}
```

4.4 heavy-light

```
struct Hld {
    vector<int> P, H, D, pos, top;

    Hld() {}
    void init(vector<vector<int>> &G) {
        int N = G.size();
        P.resize(N), H.resize(N), D.resize(N), pos.resize(N),
            top.resize(N);
        D[0] = -1, dfs(G, 0); int t = 0;
        rep(i, N) if (H[P[i]] != i) {
            int j = i;
            while (j != -1)
                { top[j] = i, pos[j] = t++; j = H[j]; }
        }
    }

    int dfs(vector<vector<int>> &G, int i) {
        int w = 1, mw = 0;
        D[i] = D[P[i]] + 1, H[i] = -1;
        for (int c : G[i]) {
            if (c == P[i]) continue;
```

```
        P[c] = i; int sw = dfs(G, c); w += sw;
        if (sw > mw) H[i] = c, mw = sw;
      }
      return w;
    }

    template <class OP>
    void path(int u, int v, OP op) {
      while (top[u] != top[v]) {
        if (D[top[u]] > D[top[v]]) swap(u, v);
        op(pos[top[v]], pos[v] + 1); v = P[top[v]];
      }
      if (D[u] > D[v]) swap(u, v);
      op(pos[u], pos[v] + 1); // value on vertex
      // op(pos[u]+1, pos[v] + 1); // value on path
    }

    // segment tree
    template <class T, class S>
    void update(S &seg, int i, T val) {
      seg.update(pos[i], val);
    }

    // segment tree lazy
    template <class T, class S>
    void update(S &seg, int u, int v, T val) {
      path(u, v, [&](int l, int r) { seg.update(l, r, val);
          });
    }

    template <class T, class S>
    T query(S &seg, int u, int v) {
      T ans = 0;
          // neutral element
      path(u, v, [&](int l, int r) { ans += seg.query(l, r)
          ; }); // query op
      return ans;
    }
};
```

4.5 hungarian

```
// find a maximum gain perfect matching in the given
    bipartite complete graph.
// input: gain matrix (G_{xy} = benefit of joining vertex x
    in set X with vertex
// y in set Y).
// output: maximum gain matching in members ‘xy[x]‘ and ‘yx[
    y]‘.
// runtime: O(N^3)
struct Hungarian {
  int N, qi, root;
  vector<vector<ll>> gain;
  vector<int> xy, yx, p, q, slackx;
  vector<ll> lx, ly, slack;
  vector<bool> S, T;

  void add(int x, int px) {
    S[x] = true, p[x] = px;
    rep(y, N) if (lx[x] + ly[y] - gain[x][y] < slack[y])
        {
      slack[y] = lx[x] + ly[y] - gain[x][y], slackx[y]
          = x;
    }
  }

  void augment(int x, int y) {
    while (x != -2) {
      yx[y] = x; swap(xy[x], y); x = p[x];
    }
  }

  void improve() {
    S.assign(N, false), T.assign(N, false), p.assign(N,
        -1);
    qi = 0, q.clear();
    rep(x, N) if (xy[x] == -1) {
      q.push_back(root = x), p[x] = -2, S[x] = true;
      break;
    }
    rep(y, N) slack[y] = lx[root] + ly[y] - gain[root][y
        ], slackx[y] = root;

    while (true) {
      while (qi < q.size()) {
        int x = q[qi++];
        rep(y, N) if (lx[x] + ly[y] == gain[x][y] && !
            T[y]) {
          if (yx[y] == -1) return augment(x, y);
          T[y] = true, q.push_back(yx[y]), add(yx[y
              ], x);
        }
      }

      ll d = INF;
      rep(y, N) if (!T[y]) d = min(d, slack[y]);
      rep(x, N) if (S[x]) lx[x] -= d;
      rep(y, N) if (T[y]) ly[y] += d;
      rep(y, N) if (!T[y]) slack[y] -= d;
      rep(y, N) if (!T[y] && slack[y] == 0) {
        if (yx[y] == -1) return augment(slackx[y], y);
        T[y] = true;
        if (!S[yx[y]]) q.push_back(yx[y]), add(yx[y],
            slackx[y]);
      }
    }
  }

  Hungarian(vector<vector<ll>> g)
    : N(g.size()), gain(g), xy(N, -1), yx(N, -1), lx(N, -
        INF),
    ly(N), slack(N), slackx(N) {
    rep(x, N) rep(y, N) lx[x] = max(lx[x], ly[y]);
    rep(i, N) improve();
  }
};
```

4.6 kuhn

```
// get a maximum cardinality matching in a bipartite graph.
// input: adjacency lists.
// output: matching (in ‘mt‘ member).
// runtime: O(V E)
struct Kuhn {
  int N, size;
  vector<vector<int>> G;
  vector<bool> seen;
  vector<int> mt;

  bool visit(int i) {
    if (seen[i]) return false;
    seen[i] = true;
    for (int to : G[i])
      if (mt[to] == -1 || visit(mt[to])) {
        mt[to] = i;
        return true;
      }
    return false;
  }

  Kuhn(vector<vector<int>> adj) : G(adj), N(G.size()), mt(N
      , -1) {
    rep(i, N) {
      seen.assign(N, false);
      size += visit(i);
    }
  }
```

```
};
```

4.7 lca

```
// calculates the lowest common ancestor for any two nodes
    in O(log N) time,
// with O(N log N) preprocessing
struct Lca {
    int L;
    vector<vector<int>> up;
    vector<pair<int, int>> time;

    Lca() {}
    void init(const vector<vector<int>> &G) {
        int N = G.size(); L = N <= 1 ? 0 : 32 - __builtin_clz
            (N - 1);
        up.resize(L + 1); rep(l, L + 1) up[l].resize(N);
        time.resize(N); int t = 0; visit(G, 0, 0, t);
        rep(l, L) rep(i, N) up[l + 1][i] = up[l][up[l][i]];
    }

    void visit(const vector<vector<int>> &G, int i, int p,
        int &t) {
        up[0][i] = p;
        time[i].first = t++;
        for (int edge : G[i]) {
            if (edge == p) continue;
            visit(G, edge, i, t);
        }
        time[i].second = t++;
    }

    bool is_anc(int up, int dn) {
        return time[up].first <= time[dn].first &&
            time[dn].second <= time[up].second;
    }

    int get(int i, int j) {
        if (is_anc(i, j)) return i;
        if (is_anc(j, i)) return j;
        int l = L;
        while (l >= 0) {
            if (is_anc(up[l][i], j)) l--;
            else i = up[l][i];
        }
        return up[0][i];
    }
};
```

4.8 maxflow-mincost

```
// untested

#include "../common.h"

const ll INF = 1e18;

struct Edge {
    int u, v;
    ll c, w, f = 0;
};

// find the minimum-cost flow among all maximum-flow flows.
//
// time: O(F V E)        F is the maximum flow
//      O(V E + F E log V) if bellman-ford is replaced by
//    johnson
struct Flow {
    int N, s, t;
    vector<vector<int>> G;
    vector<Edge> E;
    vector<ll> d;
    vector<int> p;

    Flow() {}
    Flow(int N, int s, int t) : N(N), s(s), t(t), G(N) {}

    void add_edge(int u, int v, ll c, ll w) {
        G[u].push_back(E.size());
        E.push_back({u, v, c, w});
        G[v].push_back(E.size());
        E.push_back({v, u, 0, -w});
    }

    void calcdists() {
        // replace bellman-ford with johnson for better time
        d.assign(N, INF);
        p.assign(N, -1);
        d[s] = 0;
        rep(i, N - 1) rep(ei, E.size()) {
            Edge &e = E[ei];
            ll n = d[e.u] + e.w;
            if (d[e.u] < INF && e.c - e.f > 0 && n < d[e.v])
                d[e.v] = n, p[e.v] = ei;
        }
    }

    ll maxflow() {
        ll ff = 0;
        while (true) {
            calcdists();
            if (p[t] == -1) break;

            ll f = INF;
            int cur = t;
            while (p[cur] != -1) {
                Edge &e = E[p[cur]];
                f = min(f, e.c - e.f);
                cur = e.u;
            }

            int cur = t;
            while (p[cur] != -1) {
                E[p[cur]].f += f;
                E[p[cur] ^ 1].f -= f;
            }

            ff += f;
        }
        return ff;
    }
};
```

4.9 push-relabel

```
#include "../common.h"

const ll INF = 1e18;

// maximum flow algorithm.
// to run, use `maxflow()`.
//
// time: O(V^2 sqrt(E)) <= O(V^3)
// memory: O(V^2)
struct PushRelabel {
    vector<vector<ll>> cap, flow;
    vector<ll> excess;
    vector<int> height;

    PushRelabel() {}
    void resize(int N) { cap.assign(N, vector<ll>(N)); }

    // push as much excess flow as possible from u to v.
    void push(int u, int v) {
        ll f = min(excess[u], cap[u][v] - flow[u][v]);
        flow[u][v] += f;
        flow[v][u] -= f;
        excess[v] += f;
```

```
        excess[u] -= f;
    }

    // relabel the height of a vertex so that excess flow may
        be pushed.
    void relabel(int u) {
        int d = INT32_MAX;
        rep(v, cap.size()) if (cap[u][v] - flow[u][v] > 0) d
            =
            min(d, height[v]);
        if (d < INF) height[u] = d + 1;
    }

    // get the maximum flow on the network specified by `cap`
        with source `s`
    // and sink `t`.
    // node-to-node flows are output to the `flow` member.
    ll maxflow(int s, int t) {
        int N = cap.size(), M;
        flow.assign(N, vector<ll>(N));
        height.assign(N, 0), height[s] = N;
        excess.assign(N, 0), excess[s] = INF;
        rep(i, N) if (i != s) push(s, i);

        vector<int> q;
        while (true) {
            // find the highest vertices with excess
            q.clear(), M = 0;
            rep(i, N) {
                if (excess[i] <= 0 || i == s || i == t)
                    continue;
                if (height[i] > M) q.clear(), M = height[i];
                if (height[i] >= M) q.push_back(i);
            }
            if (q.empty()) break;
            // process vertices
            for (int u : q) {
                bool relab = true;
                rep(v, N) {
                    if (excess[u] <= 0) break;
                    if (cap[u][v] - flow[u][v] > 0 && height[u
                        ] > height[v])
                        push(u, v), relab = false;
                }
                if (relab) {
                    relabel(u);
                    break;
                }
            }
        }
        ll f = 0; rep(i, N) f += flow[i][t]; return f;
    }
};
```

4.10 strongly-connected-components

```
// compute strongly connected components.
// time: O(V + E), memory: O(V)
//
// after building:
//  comp = map from vertex to component (components are
    toposorted, root first, leaf last)
//  N = number of components
//  G = condensation graph (component DAG)
//
// byproducts:
//  vgi = transposed graph
//  order = reverse topological sort (leaf first, root last)
//
// others:
//  vn = number of vertices
//  vg = original vertex graph
struct Scc {
    int vn, N;
    vector<int> order, comp;
    vector<vector<int>> vg, vgi, G;

    void toposort(int u) {
        if (comp[u]) return;
        comp[u] = -1;
        for (int v : vg[u]) toposort(v);
        order.push_back(u);
    }

    bool carve(int u) {
        if (comp[u] != -1) return false;
        comp[u] = N;
        for (int v : vgi[u]) {
            carve(v);
            if (comp[v] != N) G[comp[v]].push_back(N);
        }
        return true;
    }

    Scc() {}
    Scc(vector<vector<int>> &g) : vn(g.size()), vg(g), comp(
        vn), vgi(vn), G(vn), N(0) {
        rep(u, vn) toposort(u);
        rep(u, vn) for (int v : vg[u]) vgi[v].push_back(u);
        invrep(i, vn) N += carve(order[i]);
    }
};
```

4.11 two-sat

```
// calculate the solvability of a system of logical
    equations, where every equation is of the form `a or b
    `.
// `neg`: get negation of `u`
// `then`: `u` implies `v`
// `any`: `u` or `v`
// `set`: `u` is true
//
// after `solve` (O(V+E)) returns true, `sol` contains one
    possible solution.
// determining all solutions is O(V*E) hard (requires
    computing reachability in a DAG).
struct TwoSat {
    int N; vector<vector<int>> G;
    Scc scc; vector<bool> sol;
    TwoSat(int n) : N(n), G(2 * n), sol(n) {}
    TwoSat() {}

    int neg(int u) { return (u + N) % (2 * N); }
    void then(int u, int v) { G[u].push_back(v), G[neg(v)].
        push_back(neg(u)); }
    void any(int u, int v) { then(neg(u), v); }
    void set(int u) { G[neg(u)].push_back(u); }

    bool solve() {
        scc = Scc(G);
        rep(u, N) if (scc.comp[u] == scc.comp[neg(u)]) return
            false;
        rep(u, N) sol[u] = (scc.comp[u] > scc.comp[neg(u)]);
        return true;
    }
};
```

5 implementation

5.1 dsu

```
struct Dsu {
    vector<int> p, r;
```

```
    // initialize the disjoint-set-union to all unitary sets
    void reset(int N) {
        p.resize(N), r.assign(N, 0);
        rep(i, N) p[i] = i;
    }

    // find the leader node corresponding to node 'i'
    int find(int i) {
        if (p[i] != i) p[i] = find(p[i]);
        return p[i];
    }

    // perform union on the two sets that 'i' and 'j' belong
        to
    void unite(int i, int j) {
        i = find(i), j = find(j);
        if (i == j) return;
        if (r[i] > r[j]) swap(i, j);
        if (r[i] == r[j]) r[j] += 1;
        p[i] = j;
    }
};
```

5.2 mo

```
struct Query { int l, r, idx; };

// answer segment queries using only 'add(i)', 'remove(i)'
    and 'get()'
// functions.
//
// complexity: O((N + Q) * sqrt(N) * F)
//   N = length of the full segment
//   Q = amount of queries
//   F = complexity of the 'add', 'remove' functions
template <class A, class R, class G, class T>
void mo(vector<Query> &queries, vector<T> &ans, A add, R
    remove, G get) {
    int Q = queries.size(), B = (int)sqrt(Q);
    sort(queries.begin(), queries.end(), [&](Query &a, Query
        &b) {
        return make_pair(a.l / B, a.r) < make_pair(b.l / B, b
            .r);
    });
    ans.resize(Q);

    int l = 0, r = 0;
    for (auto &q : queries) {
        while (r < q.r) add(r), r++;
        while (l > q.l) l--, add(l);
        while (r > q.r) r--, remove(r);
        while (l < q.l) remove(l), l++;
        ans[q.idx] = get();
    }
}
```

5.3 persistent-segment-tree-lazy

```
template <class T>
struct Node {
    T x, lz;
    int l = -1, r = -1;
};

template <class T>
struct Pstl {
    int N;
    vector<Node<T>> a;
    vector<int> head;

    T qneut() { return 0; }
    T merge(T l, T r) { return l + r; }
    T uneut() { return 0; }
    T accum(T u, T x) { return u + x; }
    T apply(T x, T lz, int l, int r) { return x + (r - l) *
        lz; }

    int build(int vl, int vr) {
        if (vr - vl == 1) a.push_back({qneut(), uneut()}); //
            node construction
        else {
            int vm = (vl + vr) / 2, l = build(vl, vm), r =
                build(vm, vr);
            a.push_back({merge(a[l].x, a[r].x), uneut(), l, r
                }); // query merge
        }
        return a.size() - 1;
    }

    T query(int l, int r, int v, int vl, int vr, T acc) {
        if (l >= vr || r <= vl) return qneut();
            // query neutral
        if (l <= vl && r >= vr) return apply(a[v].x, acc, vl,
            vr); // update op
        acc = accum(acc, a[v].lz);
            // update merge
        int vm = (vl + vr) / 2;
        return merge(query(l, r, a[v].l, vl, vm, acc), query(
            l, r, a[v].r, vm, vr, acc)); // query merge
    }

    int update(int l, int r, T x, int v, int vl, int vr) {
        if (l >= vr || r <= vl || r <= l) return v;
        a.push_back(a[v]);
        v = a.size() - 1;
        if (l <= vl && r >= vr) {
            a[v].x = apply(a[v].x, x, vl, vr); // update op
            a[v].lz = accum(a[v].lz, x);    // update merge
        } else {
            int vm = (vl + vr) / 2;
            a[v].l = update(l, r, x, a[v].l, vl, vm);
            a[v].r = update(l, r, x, a[v].r, vm, vr);
            a[v].x = merge(a[a[v].l].x, a[a[v].r].x); //
                query merge
        }
        return v;
    }

    Pstl() {}
    Pstl(int N) : N(N) { head.push_back(build(0, N)); }

    T query(int t, int l, int r) {
        return query(l, r, head[t], 0, N, uneut()); // update
            neutral
    }
    int update(int t, int l, int r, T x) {
        return head.push_back(update(l, r, x, head[t], 0, N))
            , head.size() - 1;
    }
};
```

5.4 persistent-segment-tree

```
// usage:
// Pst<Node<ll>> pst;
// pst = {N};
// int newtime = pst.update(time, index, value);
// Node<ll> result = pst.query(newtime, left, right);

template <class T>
struct Node {
    T x;
    int l = -1, r = -1;

    Node() : x(0) {}
    Node(T x) : x(x) {}
```

```
    Node(Node a, Node b, int l = -1, int r = -1) : x(a.x + b.
        x), l(l), r(r) {}
};

template <class U>
struct Pst {
    int N;
    vector<U> a;
    vector<int> head;

    int build(int vl, int vr) {
        if (vr - vl == 1) a.push_back(U()); // node
            construction
        else {
            int vm = (vl + vr) / 2, l = build(vl, vm), r =
                build(vm, vr);
            a.push_back(U(a[l], a[r], l, r)); // query merge
        }
        return a.size() - 1;
    }

    U query(int l, int r, int v, int vl, int vr) {
        if (l >= vr || r <= vl) return U(); // query neutral
        if (l <= vl && r >= vr) return a[v];
        int vm = (vl + vr) / 2;
        return U(query(l, r, a[v].l, vl, vm), query(l, r, a[v
            ].r, vm, vr)); // query merge
    }

    int update(int i, U x, int v, int vl, int vr) {
        a.push_back(a[v]);
        v = a.size() - 1;
        if (vr - vl == 1) a[v] = x; // update op
        else {
            int vm = (vl + vr) / 2;
            if (i < vm) a[v].l = update(i, x, a[v].l, vl, vm)
                ;
            else a[v].r = update(i, x, a[v].r, vm, vr);
            a[v] = U(a[a[v].l], a[a[v].r], a[v].l, a[v].r);
                // query merge
        }
        return v;
    }

    Pst() {}
    Pst(int N) : N(N) { head.push_back(build(0, N)); }

    U query(int t, int l, int r) {
        return query(l, r, head[t], 0, N);
    }
    int update(int t, int i, U x) {
        return head.push_back(update(i, x, head[t], 0, N)),
            head.size() - 1;
    }
};
```

5.5 segment-tree-lazy

```
// 0-based, inclusive-exclusive
// usage:
// Stl3<ll> a;
// a = {N};
template <class T>
struct Stl3 {
    // immediate, lazy
    vector<pair<T, T>> a;

    T qneutral() { return 0; }
    T merge(T l, T r) { return l + r; }
    T uneutral() { return 0; }
    void update(pair<T, T> &u, T val, int l, int r) { u.first
        += val * (r - l), u.second += val; }

    Stl3() {}
    Stl3(int N) : a(4 * N, {qneutral(), uneutral()}) {} //
        node neutral

    void push(int v, int vl, int vm, int vr) {
        update(a[2 * v], a[v].second, vl, vm); // node update
        update(a[2 * v + 1], a[v].second, vm, vr); // node
            update
        a[v].second = uneutral();               // update
            neutral
    }

    // query for range [l, r)
    T query(int l, int r, int v = 1, int vl = 0, int vr = -1)
        {
        if (vr == -1) vr = a.size() / 4;
        if (l <= vl && r >= vr) return a[v].first; // query
            op
        if (l >= vr || r <= vl) return qneutral(); // query
            neutral
        int vm = (vl + vr) / 2;
        push(v, vl, vm, vr);
        return merge(query(l, r, 2 * v, vl, vm), query(l, r,
            2 * v + 1, vm, vr)); // item merge
    }

    // update range [l, r) using val
    void update(int l, int r, T val, int v = 1, int vl = 0,
        int vr = -1) {
        if (vr == -1) vr = a.size() / 4;
        if (l >= vr || r <= vl || r <= l) return;
        if (l <= vl && r >= vr) update(a[v], val, vl, vr); //
            node update
        else {
            int vm = (vl + vr) / 2;
            push(v, vl, vm, vr);
            update(l, r, val, 2 * v, vl, vm);
            update(l, r, val, 2 * v + 1, vm, vr);
            a[v].first = merge(a[2 * v].first, a[2 * v + 1].
                first); // node merge
        }
    }
};

struct Node {
    ll x, lazy;

    Node() : x(neutral()), lazy(0) {} // query neutral,
        update neutral
    Node(ll x_) : Node() { x = x_; }
    Node(Node &l, Node &r) : Node() { refresh(l, r); } //
        node merge construction
    void refresh(Node &l, Node &r) { x = merge(l.x, r.x); }
        // node merge

    void update(ll val, int l, int r) { x += val * (r - l),
        lazy += val; } // update-query, update accumulate
    ll take() {
        ll z = 0; // update neutral
        swap(lazy, z);
        return z;
    }

    ll query() { return x; }
    static ll neutral() { return 0; }          // query
        neutral
    static ll merge(ll l, ll r) { return l + r; } // query
        merge
};

template <class T, class Node>
struct Stl {
    vector<Node> node;

    void reset(int N) { node.assign(4 * N, {}); } // node
        neutral
```

```cpp
void build(const vector<T> &a, int v = 1, int vl = 0, int
     vr = -1) {
    node.resize(4 * a.size()), vr = vr == -1 ? node.size
        () / 4 : vr;
    if (vr - vl == 1) {
        node[v] = {a[vl]}; // node construction
        return;
    }
    int vm = (vl + vr) / 2;
    build(a, 2 * v, vl, vm);
    build(a, 2 * v + 1, vm, vr);
    node[v] = {node[2 * v], node[2 * v + 1]}; // node
        merge construction
}

void push(int v, int vl, int vm, int vr) {
    T lazy = node[v].take();          // update neutral
    node[2 * v].update(lazy, vl, vm); // node update
    node[2 * v + 1].update(lazy, vm, vr); // node update
}

// query for range [l, r)
T query(int l, int r, int v = 1, int vl = 0, int vr = -1)
     {
    if (vr == -1) vr = node.size() / 4;
    if (l <= vl && r >= vr) return node[v].query(); //
        query op
    if (l >= vr || r <= vl) return Node::neutral(); //
        query neutral
    int vm = (vl + vr) / 2;
    push(v, vl, vm, vr);
    return Node::merge(query(l, r, 2 * v, vl, vm), query(
        l, r, 2 * v + 1, vm, vr)); // item merge
}

// update range [l, r) using val
void update(int l, int r, T val, int v = 1, int vl = 0,
     int vr = -1) {
    if (vr == -1) vr = node.size() / 4;
    if (l >= vr || r <= vl || r <= l) return;
    if (l <= vl && r >= vr) node[v].update(val, vl, vr);
        // node update
    else {
        int vm = (vl + vr) / 2;
        push(v, vl, vm, vr);
        update(l, r, val, 2 * v, vl, vm);
        update(l, r, val, 2 * v + 1, vm, vr);
        node[v].refresh(node[2 * v], node[2 * v + 1]); //
             node merge
        }
    }
};
```

5.6 segment-tree

```cpp
// usage:
// St<Node<ll>> st;
// st = {N};
// st.update(index, new_value);
// Node<ll> result = st.query(left, right);

template <class T>
struct Node {
    T x;
    Node() : x(0) {}
    Node(T x) : x(x) {}
    Node(Node a, Node b) : x(a.x + b.x) {}
};

template <class U>
struct St {
    vector<U> a;

    St() {}
    St(int N) : a(4 * N, U()) {} // node neutral

    // query for range [l, r)
    U query(int l, int r, int v = 1, int vl = 0, int vr = -1)
         {
        if (vr == -1) vr = a.size() / 4;
        if (l <= vl && r >= vr) return a[v]; // item
            construction
        int vm = (vl + vr) / 2;
        if (l >= vr || r <= vl) return U();
                                        // item neutral
        return U(query(l, r, 2 * v, vl, vm), query(l, r, 2 *
            v + 1, vm, vr)); // item merge
    }

    // set element i to val
    void update(int i, U val, int v = 1, int vl = 0, int vr =
         -1) {
        if (vr == -1) vr = a.size() / 4;
        if (vr - vl == 1) a[v] = val; // item update
        else {
            int vm = (vl + vr) / 2;
            if (i < vm) update(i, val, 2 * v, vl, vm);
            else update(i, val, 2 * v + 1, vm, vr);
            a[v] = U(a[2 * v], a[2 * v + 1]); // node merge
        }
    }
};
```

5.7 sparse-table

```cpp
// handle immutable range maximum queries (or any idempotent
     query) in O(1)
template <class T>
struct Sparse {
    vector<vector<T>> st;

    T op(T a, T b) { return max(a, b); }

    Sparse() {}

    void reset(int N) { st = {vector<T>(N)}; }
    void set(int i, T val) { st[0][i] = val; }

    // O(N log N) time
    // O(N log N) memory
    void init() {
        int N = st[0].size();
        int npot = N <= 1 ? 1 : 32 - __builtin_clz(N);
        st.resize(npot);
        repx(i, 1, npot) rep(j, N + 1 - (1 << i)) st[i].
            push_back(
          op(st[i - 1][j], st[i - 1][j + (1 << (i - 1))]));
                // query op
    }

    // query maximum in the range [l, r) in O(1) time
    // range must be nonempty!
    T query(int l, int r) {
        int i = 31 - __builtin_clz(r - l);
        return op(st[i][l], st[i][r - (1 << i)]); // query op
    }
};
```

5.8 unordered-map

```cpp
// hackproof rng
static mt19937 rng(chrono::steady_clock::now().
    time_since_epoch().count());

// deterministic rng
```

```
uint64_t splitmix64(uint64_t *x) {
    uint64_t z = (*x += 0x9e3779b97f4a7c15);
    z = (z ^ (z >> 30)) * 0xbf58476d1ce4e5b9;
    z = (z ^ (z >> 27)) * 0x94d049bb133111eb;
    return z ^ (z >> 31);
}

// hackproof unordered map hash
struct Hash {
    size_t operator()(const ll &x) const {
        static const uint64_t RAND =
            chrono::steady_clock::now().time_since_epoch().
                count();
        uint64_t z = x + RAND + 0x9e3779b97f4a7c15;
        z = (z ^ (z >> 30)) * 0xbf58476d1ce4e5b9;
        z = (z ^ (z >> 27)) * 0x94d049bb133111eb;
        return z ^ (z >> 31);
    }
};

// hackproof unordered_map
template <class T, class U>
using umap = unordered_map<T, U, Hash>;

// hackproof unordered_set
template <class T>
using uset = unordered_set<T, Hash>;
```

6 imprimible

7 math

7.1 arithmetic

```
// floor(log2(n)) without precision loss
inline int floor_log2(int n) { return n <= 1 ? 0 : 31 -
    __builtin_clz(n); }
// ceil(log2(n)) without precision loss
inline int ceil_log2(int n) { return n <= 1 ? 0 : 32 -
    __builtin_clz(n - 1); }

inline ll floordiv(ll a, ll b) {
    return a / b - ((a ^ b) < 0 && a % b);
}

inline ll ceildiv(ll a, ll b) {
    return a / b + ((a ^ b) >= 0 && a % b);
}

// a^e through binary exponentiation.
ll binexp(ll a, ll e) {
    ll res = 1; // neutral element
    while (e) {
        if (e & 1) res = res * a; // multiplication
        a = a * a;                // multiplication
        e >>= 1;
    }
    return res;
}
```

7.2 crt

```
pair<ll, ll> solve_crt(const vector<pair<ll, ll>> &eqs) {
    ll a0 = eqs[0].first, p0 = eqs[0].second;
    repx(i, 1, eqs.size()) {
        ll a1 = eqs[i].first, p1 = eqs[i].second;
        ll k1, k0;
        ll d = ext_gcd(p1, p0, k1, k0);
        a0 -= a1;
        if (a0 % d != 0) return {-1, -1};
        p0 = p0 / d * p1;
        a0 = a0 / d * k1 % p0 * p1 % p0 + a1;
        a0 = (a0 % p0 + p0) % p0;
    }
    return {a0, p0};
}
```

7.3 discrete-log

```
// discrete logarithm log_a(b).
// solve b ^ x = a (mod M) for the smallest x.
// returns -1 if no solution is found.
//
// time: O(sqrt(M))
ll dlog(ll a, ll b, ll M) {
    ll k = 1, s = 0;
    while (true) {
        ll g = __gcd(b, M);
        if (g <= 1) break;
        if (a == k) return s;
        if (a % g != 0) return -1;
        a /= g, M /= g, s += 1, k = b / g * k % M;
    }
    ll N = sqrt(M) + 1;

    umap<ll, ll> r;
    rep(q, N + 1) {
        r[a] = q;
        a = a * b % M;
    }

    ll bN = binexp(b, N, M), bNp = k;
    repx(p, 1, N + 1) {
        bNp = bNp * bN % M;
        if (r.count(bNp)) return N * p - r[bNp] + s;
    }
    return -1;
}
```

7.4 gauss

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

// solve a system of equations.
// complexity: O(min(N, M) * N * M)
//
// `a` is a list of rows
// the last value in each row is the result of the equation
// return values:
//  0 -> no solutions
//  1 -> unique solution, stored in `ans`
// -1 -> infinitely many solutions, one of which is stored
//    in `ans`
// UNTESTED
int gauss(vector<vector<double>> a, vector<double> &ans) {
    int N = a.size(), M = a[0].size() - 1;

    vector<int> where(M, -1);
    for (int j = 0, i = 0; j < M && i < N; j++) {
        int sel = i;
        repx(k, i, N) if (abs(a[k][j]) > abs(a[sel][j])) sel
            = k;
        if (abs(a[sel][j]) < EPS) continue;
        repx(k, j, M + 1) swap(a[sel][k], a[i][k]);
        where[j] = i;

        rep(k, N) if (k != i) {
            double c = a[k][j] / a[i][j];
            repx(l, j, M + 1) a[k][l] -= a[i][l] * c;
        }
        i++;
    }
```

```
    ans.assign(M, 0);
    rep(i, M) if (where[i] != -1) ans[i] = a[where[i]][M] / a
        [where[i]][i];
    rep(i, N) {
        double sum = 0;
        rep(j, M) sum += ans[j] * a[i][j];
        if (abs(sum - a[i][M]) > EPS) return 0;
    }

    rep(i, M) if (where[i] == -1) return -1;
    return 1;
}
```

7.5 matrix

```
using T = ll;
struct Mat {
    int N, M;
    vector<vector<T>> v;

    Mat(int n, int m) : N(n), M(m), v(N, vector<T>(M)) {}
    Mat(int n) : Mat(n, n) { rep(i, N) v[i][i] = 1; }

    vector<T> &operator[](int i) { return v[i]; }

    Mat operator*(Mat &r) {
        assert(M == r.N);
        int n = N, m = r.M, p = M;
        Mat a(n, m);
        rep(i, n) rep(j, m) {
            a[i][j] = T();                              //
                neutral
            rep(k, p) a[i][j] = a[i][j] + v[i][k] * r[k][j];
                // mul, add
        }
        return a;
    }

    Mat binexp(ll e) {
        assert(N == M);
        Mat a = *this, res(N); // neutral
        while (e) {
            if (e & 1) res = res * a; // mul
            a = a * a;                // mul
            e >>= 1;
        }
        return res;
    }

    friend ostream &operator<<(ostream &s, Mat &a) {
        rep(i, a.N) {
            rep(j, a.M) s << a[i][j] << " ";
            s << endl;
        }
        return s;
    }
};
```

7.6 mod

```
ll binexp(ll a, ll e, ll M) {
    assert(e >= 0);
    ll res = 1 % M;
    while (e) {
        if (e & 1) res = res * a % M;
        a = a * a % M;
        e >>= 1;
    }
    return res;
}

ll multinv(ll a, ll M) { return binexp(a, M - 2, M); }

// calculate gcd(a, b).
// also, calculate x and y such that:
// a * x + b * y == gcd(a, b)
//
// time: O(log min(a, b))
// (ignoring complexity of arithmetic)
ll ext_gcd(ll a, ll b, ll &x, ll &y) {
    if (b == 0) {
        x = 1, y = 0;
        return a;
    }
    ll d = ext_gcd(b, a % b, y, x);
    y -= a / b * x;
    return d;
}

// compute inverse with any M.
// a and M must be coprime for inverse to exist!
ll multinv_euc(ll a, ll M) {
    ll x, y;
    ext_gcd(a, M, x, y);
    return x;
}

// multiply two big numbers (~10^18) under a large modulo,
    without resorting to
// bigints.
ll bigmul(ll x, ll y, ll M) {
    ll z = 0;
    while (y) {
        if (y & 1) z = (z + x) % M;
        x = (x << 1) % M, y >>= 1;
    }
    return z;
}

// all modular inverses from 1 to inv.size()-1
void multinv_all(vector<ll> &inv) {
    inv[1] = 1;
    repx(i, 2, inv.size())
        inv[i] = MOD - (MOD / i) * inv[MOD % i] % MOD;
}

struct Mod {
    int a;
    static const int M = 1e9 + 7;

    Mod(ll aa) : a((aa % M + M) % M) {}

    Mod operator+(Mod rhs) const { return (a + rhs.a) % M; }
    Mod operator-(Mod rhs) const { return (a - rhs.a + M) % M
        ; }
    Mod operator-() const { return Mod(0) - *this; }
    Mod operator*(Mod rhs) const { return (ll)a * rhs.a % M;
        }

    Mod operator+=(Mod rhs) { return *this = *this + rhs; }
    Mod operator-=(Mod rhs) { return *this = *this - rhs; }
    Mod operator*=(Mod rhs) { return *this = *this * rhs; }

    Mod bigmul(ll big) const { return ::bigmul(a, big, M); }

    Mod binexp(ll e) const { return ::binexp(a, e, M); }
    // Mod multinv() const { return ::multinv(a, M); } //
        prime M
    Mod multinv() const { return ::multinv_euc(a, M); } //
        possibly composite M
};

// dynamic modulus
struct DMod {
    int a, M;

    DMod(ll aa, ll m) : M(m), a((aa % m + m) % m) {}
```

```
    DMod operator+(DMod rhs) const { return {(a + rhs.a) % M,
         M}; }
    DMod operator-(DMod rhs) const { return {(a - rhs.a + M)
        % M, M}; }
    DMod operator-() const { return DMod(0, M) - *this; }
    DMod operator*(DMod rhs) const { return {(ll)a * rhs.a %
        M, M}; }

    DMod operator+=(DMod rhs) { return *this = *this + rhs; }
    DMod operator-=(DMod rhs) { return *this = *this - rhs; }
    DMod operator*=(DMod rhs) { return *this = *this * rhs; }

    DMod bigmul(ll big) const { return {::bigmul(a, big, M),
        M}; }

    DMod binexp(ll e) const { return {::binexp(a, e, M), M};
        }
    DMod multinv() const { return {::multinv(a, M), M}; } //
        prime M
    // DMod multinv() const { return {::multinv_euc(a, M), M
        }; } // possibly composite M
};
```

7.7 poly

```
using cd = complex<double>;
const double PI = acos(-1);

// compute the DFT of a power-of-two-length sequence.
// if `inv` is true, computes the inverse DFT.
//
// the DFT of a polynomial A(x) = A0 + A1*x + A2*x^2 + ... +
     An*x^n is the array
// of the polynomial A evaluated in all nths roots of unity:
     [A(w0), A(w1),
// A(w2), ..., A(wn-1)], where w0 = 1 and w1 is the nth
     principal root of unity.
void fft(vector<cd> &a, bool inv) {
    int N = a.size(), k = 0;
    assert(N == 1 << __builtin_ctz(N));

    rep(i, N) {
        int b = N >> 1;
        while (k & b) k ^= b, b >>= 1;
        k ^= b;
        if (i < k) swap(a[i], a[k]);
    }

    for (int l = 2; l <= N; l <<= 1) {
        double ang = 2 * PI / l * (inv ? -1 : 1);
        cd wl(cos(ang), sin(ang));
        for (int i = 0; i < N; i += l) {
            cd w(1);
            repx(j, 0, l / 2) {
                cd u = a[i + j], v = a[i + j + l / 2] * w;
                a[i + j] = u + v;
                a[i + j + l / 2] = u - v;
                w *= wl;
            }
        }
    }

    if (inv)
        for (cd &x : a) x /= N;
}

const ll MOD = 7340033, ROOT = 5, ROOTPOW = 1 << 20;

void find_root_of_unity(ll M) {
    ll c = M - 1, k = 0;
    while (c % 2 == 0) c /= 2, k += 1;

    // find proper divisors of M - 1
    vector<int> divs;
    repx(d, 1, c) {
        if (d * d > c) break;
        if (c % d == 0) rep(i, k + 1) divs.push_back(d << i);
    }
    rep(i, k) divs.push_back(c << i);

    // find any primitive root of M
    ll G = -1;
    repx(g, 2, M) {
        bool ok = true;
        for (int d : divs) ok &= (binexp(g, d, M) != 1);
        if (ok) {
            G = g;
            break;
        }
    }
    assert(G != -1);

    ll w = binexp(G, c, M);
    cerr << M << " = c * 2^k + 1" << endl;
    cerr << " c = " << c << endl;
    cerr << " k = " << k << endl;
    cerr << "w^(2^k) == 1" << endl;
    cerr << " w = " << w << endl;
}

// compute the DFT of a power-of-two-length sequence, modulo
     a special prime
// number with principal root.
//
// the modulus _must_ be a prime number with an Nth root of
     unity, where N is a
// power of two. the FFT can only be performed on arrays of
     size <= N.
void ntt(vector<ll> &a, bool inv) {
    int N = a.size(), k = 0;
    assert(N == 1 << __builtin_ctz(N) && N <= ROOTPOW);
    rep(i, N) a[i] = (a[i] % MOD + MOD) % MOD;

    repx(i, 1, N) {
        int b = N >> 1;
        while (k & b) k ^= b, b >>= 1;
        k ^= b;
        if (i < k) swap(a[i], a[k]);
    }

    for (int l = 2; l <= N; l <<= 1) {
        ll wl = inv ? multinv(ROOT, MOD) : ROOT;
        for (ll i = ROOTPOW; i > l; i >>= 1) wl = wl * wl %
            MOD;
        for (int i = 0; i < N; i += l) {
            ll w = 1;
            repx(j, 0, l / 2) {
                ll u = a[i + j], v = a[i + j + l / 2] * w %
                    MOD;
                a[i + j] = (u + v) % MOD;
                a[i + j + l / 2] = (u - v + MOD) % MOD;
                w = w * wl % MOD;
            }
        }
    }

    ll ninv = multinv(N, MOD);
    if (inv)
        for (ll &x : a) x = x * ninv % MOD;
}

void convolve(vector<ll> &a, vector<ll> b, int n) {
    n = 1 << (32 - __builtin_clz(2 * n - 1));
    a.resize(n), b.resize(n);
    ntt(a, false), ntt(b, false);
    rep(i, n) a[i] *= b[i];
    ntt(a, true), ntt(b, true);
}
```

```
using T = ll;
T pmul(T a, T b) { return a * b % MOD; }
T padd(T a, T b) { return (a + b) % MOD; }
T psub(T a, T b) { return (a - b + MOD) % MOD; }
T pinv(T a) { return multinv(a, MOD); }

struct Poly {
    vector<T> a;

    Poly() {}
    Poly(T c) : a(c) { trim(); }
    Poly(vector<T> c) : a(c) { trim(); }

    void trim() {
        while (!a.empty() && a.back() == 0) a.pop_back();
    }
    int deg() const { return a.empty() ? -1000000 : a.size()
         - 1; }
    Poly sub(int l, int r) const {
        r = min(r, (int)a.size()), l = min(l, r);
        return vector<T>(a.begin() + l, a.begin() + r);
    }
    Poly trunc(int n) const { return sub(0, n); }
    Poly shl(int n) const {
        Poly out = *this;
        out.a.insert(out.a.begin(), n, 0);
        return out;
    }
    Poly rev(int n, bool r = false) const {
        Poly out(*this);
        if (r) out.a.resize(max(n, (int)a.size()));
        reverse(out.a.begin(), out.a.end());
        return out.trunc(n);
    }

    Poly &operator+=(const Poly &rhs) {
        auto &b = rhs.a;
        a.resize(max(a.size(), b.size()));
        rep(i, b.size()) a[i] = padd(a[i], b[i]); // add
        trim();
        return *this;
    }
    Poly &operator-=(const Poly &rhs) {
        auto &b = rhs.a;
        a.resize(max(a.size(), b.size()));
        rep(i, b.size()) a[i] = psub(a[i], b[i]); // sub
        trim();
        return *this;
    }
    Poly &operator*=(const Poly &rhs) {
        int n = deg() + rhs.deg() + 1;
        if (n <= 0) return *this = Poly();
        n = 1 << (n <= 1 ? 0 : 32 - __builtin_clz(n - 1));
        vector<T> b = rhs.a;
        a.resize(n), b.resize(n);
        ntt(a, false), ntt(b, false);         // fft
        rep(i, a.size()) a[i] = pmul(a[i], b[i]); // mul
        ntt(a, true), trim();                 // invfft
        return *this;
    }
    Poly inv(int n) const {
        assert(deg() >= 0);
        Poly ans = pinv(a[0]); // inverse
        int b = 1;
        while (b < n) {
            Poly C = (ans * trunc(2 * b)).sub(b, 2 * b);
            ans -= (ans * C).trunc(b).shl(b);
            b *= 2;
        }
        return ans.trunc(n);
    }

    Poly operator+(const Poly &rhs) const { return Poly(*this
        ) += rhs; }
    Poly operator-(const Poly &rhs) const { return Poly(*this
        ) -= rhs; }
    Poly operator*(const Poly &rhs) const { return Poly(*this
        ) *= rhs; }

    pair<Poly, Poly> divmod(const Poly &b) const {
        if (deg() < b.deg()) return {Poly(), *this};
        int d = deg() - b.deg() + 1;
        Poly D = (rev(d) * b.rev(d).inv(d)).trunc(d).rev(d,
             true);
        return {D, *this - D * b};
    }
    Poly operator/(const Poly &b) const { return divmod(b).
        first; }
    Poly operator%(const Poly &b) const { return divmod(b).
        second; }
    Poly &operator/=(const Poly &b) { return *this = divmod(b
        ).first; }
    Poly &operator%=(const Poly &b) { return *this = divmod(b
        ).second; }

    T eval(T x) {
        T y = 0;
        invrep(i, a.size()) y = padd(pmul(y, x), a[i]); //
             add, mul
        return y;
    }
    Poly &build(vector<Poly> &tree, vector<T> &x, int v, int
         l, int r) {
        if (l == r) return tree[v] = vector<T>{-x[l], 1};
        int m = (l + r) / 2;
        return tree[v] = build(tree, x, 2 * v, l, m) *
                         build(tree, x, 2 * v + 1, m + 1, r);
    }
    void subeval(vector<Poly> &tree, vector<T> &x, vector<T>
         &y, int v, int l,
               int r) {
        if (l == r) {
            y[l] = eval(x[l]);
            return;
        }
        int m = (l + r) / 2;
        (*this % tree[2 * v]).subeval(tree, x, y, 2 * v, l, m
            );
        (*this % tree[2 * v + 1]).subeval(tree, x, y, 2 * v +
             1, m + 1, r);
    }
    // evaluate m points in O(k (log k)^2) with k = max(n, m)
        .
    vector<T> multieval(vector<T> &x) {
        int N = x.size();
        if (deg() < 0) return vector<T>(N, 0);
        vector<Poly> tree(4 * N);
        build(tree, x, 1, 0, N - 1);
        vector<T> y(N);
        subeval(tree, x, y, 1, 0, N - 1);
        return y;
    }
};
```

7.8 primes

```
// counts the divisors of a positive integer in O(sqrt(n))
ll count_divisors(ll x) {
    ll divs = 1, i = 2;
    for (ll divs = 1, i = 2; x > 1; i++) {
        if (i * i > x) {
            divs *= 2;
            break;
        }
        for (ll d = divs; x % i == 0; x /= i) divs += d;
    }
    return divs;
}
```

```cpp
// gets the prime factorization of a number in O(sqrt(n))
vector<pair<ll, int>> factorize(ll x) {
    vector<pair<ll, int>> f;
    for (ll k = 2; x > 1; k++) {
        if (k * k > x) {
            f.push_back({x, 1});
            break;
        }
        int n = 0;
        while (x % k == 0) x /= k, n++;
        if (n > 0) f.push_back({k, n});
    }
    return f;
}

// iterate over all divisors of a number.
//
// divisor count upper bound: n^(1.07 / ln ln n)
template <class OP>
void divisors(ll x, OP op) {
    auto facts = factorize(x);
    vector<int> f(facts.size());
    while (true) {
        ll y = 1;
        rep(i, f.size()) rep(j, f[i]) y *= facts[i].first;
        op(y);

        int i;
        for (i = 0; i < f.size(); i++) {
            f[i] += 1;
            if (f[i] <= facts[i].second) break;
        }
        if (i == f.size()) break;
    }
}

// computes euler totative function phi(x), counting the
    amount of integers in
// [1, x] that are coprime with x.
//
// time: O(sqrt(x))
ll phi(ll x) {
    ll phi = 1, k = 2;
    for (; x > 1; k++) {
        if (k * k > x) {
            phi *= x - 1;
            break;
        }
        ll k1 = 1, k0 = 0;
        while (x % k == 0) x /= k, k0 = k1, k1 *= k;
        phi *= k1 - k0;
    }
    return phi;
}

// computes primality up to N.
// considers 0 and 1 prime.
// O(N log N)
void sieve(int N, vector<bool> &prime) {
    prime.assign(N + 1, true);
    repx(n, 2, N + 1) if (prime[n]) for (int k = 2 * n; k <=
        N; k += n) prime[k] = false;
}
```

7.9 theorems

```cpp
// Burnside lemma
//
//    For a set X, with members x in X, and a group G, with
    operations g in G, where g(x): X -> X.
//    F_g is the set of x which are fixed points of g (ie. {
    x in X / g(x) = x }).
//    The number of orbits (connected components in the
    graph formed by assigning each x a node and
//    a directed edge between x and g(x) for every g) is
    called M.
//    M = the average of the fixed points of all g = (|F_g1|
    + |F_g2| + ... + |F_gn|) / |G|
//
//    If x are images and g are simmetries, then M
    corresponds to the amount of objects, |G|
//    corresponds to the amount of simmetries, and F_g
    corresponds to the amount of simmetrical
//    images under the simmetry g.
//
// Rational root theorem
//
//    All rational roots of the polynomials with integer
    coefficients:
//
//    a0 * x^0 + a1 * x^1 + a2 * x^2 + ... + an * x^n = 0
//
//    If these roots are represented as p / q, with p and q
    coprime,
//    - p is an integer factor of a0
//    - q is an integer factor of an
//
//    Note that if a0 = 0, then x = 0 is a root, the
    polynomial can be divided by x and the theorem
//    applies once again.
//
// Legendre's formula
//
//    Considering a prime p, the largest power p^k that
    divides n! is given by:
//
//    k = floor(n/p) + floor(n/p^2) + floor(n/p^3) + ...
//
//    Which can be computed in O(log n / log p) time
```