

Contents

1. Suffix Tree	2
2. Suffix Array	3
3. Suffix automaton	5
4. Kasai	5
5. Z-function	6
6. prefix-function	6
7. Aho	6
8. Hungarian	8
9. Hopcroft-Karp	9
10. CHT	10
11. FFT with prime mod	11
12. FFT with ld	13
13. Convex-Hull	15
14. Palindrome-Tree	16
15. STL TREE, HASH_MAP	16
16. Dinic	17
17. MCMF	19
18. Pollard	22

Suffix Tree

```

1 // Ukkonen's algorithm  $O(n)$ 
2 const int A = 27; // Alphabet size
3 struct SuffixTree {
4     struct Node { // [l, r) !!!
5         int l, r, link, par;
6         int nxt[A];
7         Node(): l(-1), r(-1), link(-1), par(-1) { fill(nxt, nxt + A, -1); }
8         Node(int _l, int _r, int _link, int _par):
9             l(_l), r(_r), link(_link), par(_par) { fill(nxt, nxt + A, -1); }
10        int &next(int c) { return nxt[c]; }
11        int get_len() const { return r - l; }
12    };
13    struct State { int v, len; };
14    vec< Node > t;
15    State cur_state;
16    vec< int > s;
17    SuffixTree(): cur_state({0, 0}) { t.push_back(Node()); }
18    // v -> v + s[l, r) !!!
19    State go(State st, int l, int r) {
20        while(l < r) {
21            if(st.len == t[st.v].get_len()) {
22                State nx = State({ t[st.v].next(s[l]), 0 });
23                if(nx.v == -1) return nx;
24                st = nx;
25                continue;
26            }
27            if(s[ t[st.v].l + st.len ] != s[l]) return State({-1, -1});
28            if(r - l < t[st.v].get_len() - st.len)
29                return State({st.v, st.len + r - l});
30            l += t[st.v].get_len() - st.len;
31            st.len = t[st.v].get_len();
32        }
33        return st;
34    }
35    int get_vertex(State st) {
36        if(t[st.v].get_len() == st.len) return st.v;
37        if(st.len == 0) return t[st.v].par;
38        Node &v = t[st.v];
39        Node &pv = t[v.par];
40        Node add(v.l, v.l + st.len, -1, v.par);
41        // nxt
42        pv.next(s[v.l]) = (int)t.size();
43        add.next(s[v.l + st.len]) = st.v;
44        // par

```

```

45     v.par = (int)t.size();
46     // [l, r)
47     v.l += st.len;
48     t.push_back(add); // !!!
49     return (int)t.size() - 1;
50 }
51 int get_link(int v) {
52     if(t[v].link != -1) return t[v].link;
53     if(t[v].par == -1) return 0;
54     int to = get_link(t[v].par);
55     to = get_vertex(
56         go(State({to, t[to].get_len()}), t[v].l + (t[v].par == 0), t[v].r)
57     );
58     return t[v].link = to;
59 }
60 void add_symbol(int c) {
61     assert(0 <= c && c < A);
62     s.push_back(c);
63     while(1) {
64         State hlp = go( cur_state, (int)s.size() - 1, (int)s.size() );
65         if(hlp.v != -1) { cur_state = hlp; break; }
66         int v = get_vertex(cur_state);
67         Node add((int)s.size() - 1, +inf, -1, v);
68         t.push_back(add);
69         t[v].next(c) = (int)t.size() - 1;
70         cur_state.v = get_link(v);
71         cur_state.len = t[cur_state.v].get_len();
72         if(!v) break;
73     }
74 }
75 };

```

Suffix Array

```

1  const int LOG = 21;
2  struct SuffixArray {
3      string s;
4      int n;
5      vec< int > p;
6      vec< int > c[LOG];
7      SuffixArray(): n(0) { }
8      SuffixArray(string ss): s(ss) {
9          s.push_back(0);
10         n = (int)s.size();
11         vec< int > pn, cn;

```

```

12     vec< int > cnt;
13     p.resize(n);
14     for(int i = 0; i < LOG; i++) c[i].resize(n);
15     pn.resize(n);
16     cn.resize(n);
17     cnt.assign(300, 0);
18     for(int i = 0; i < n; i++) cnt[s[i]]++;
19     for(int i = 1; i < (int)cnt.size(); i++) cnt[i] += cnt[i - 1];
20     for(int i = n - 1; i >= 0; i--) p[--cnt[s[i]]] = i;
21     for(int i = 1; i < n; i++) {
22         c[0][p[i]] = c[0][p[i - 1]];
23         if(s[p[i]] != s[p[i - 1]]) c[0][p[i]]++;
24     }
25     for(int lg = 0, k = 1; k < n; k <= 1, lg++) {
26         for(int i = 0; i < n; i++) {
27             if((pn[i] = p[i] - k) < 0) pn[i] += n;
28         }
29         cnt.assign(n, 0);
30         for(int i = 0; i < n; i++) cnt[c[lg][pn[i]]]++;
31         for(int i = 1; i < (int)cnt.size(); i++) cnt[i] += cnt[i - 1];
32         for(int i = n - 1; i >= 0; i--) p[--cnt[c[lg][pn[i]]]] = pn[i];
33         for(int l1, r1, l2, r2, i = 1; i < n; i++) {
34             cn[p[i]] = cn[p[i - 1]];
35             l1 = p[i - 1];
36             l2 = p[i];
37             if((r1 = l1 + k) >= n) r1 -= n;
38             if((r2 = l2 + k) >= n) r2 -= n;
39             if(c[lg][l1] != c[lg][l2] || c[lg][r1] != c[lg][r2]) cn[p[i]]++;
40         }
41         c[lg + 1] = cn;
42     }
43     p.erase(p.begin(), p.begin() + 1);
44     n--;
45 }
46 int get_lcp(int i, int j) {
47     int res = 0;
48     for(int lg = LOG - 1; lg >= 0; lg--) {
49         if(i + (1 << lg) > n || j + (1 << lg) > n) continue;
50         if(c[lg][i] == c[lg][j]) {
51             i += (1 << lg);
52             j += (1 << lg);
53             res += (1 << lg);
54         }
55     }
56     return res;
57 }

```

```
};
```

Suffix automaton

```

1 struct suf_auto {
2     vector<int> base, suf, len;
3     vector<vector<int>> g;
4     int last, sz;
5     suf_auto(): base(26, -1), g(1, base), suf(1, -1), len(1, 0), last(0), sz(1){}
6     void add_string(const string &s) {
7         for (char c : s) {
8             c -= 'a';
9             int cur = last;
10            last = sz++;
11            g.push_back(base);
12            suf.emplace_back();
13            len.push_back(len[cur] + 1);
14            while (cur != -1 && g[cur][c] == -1)
15                g[cur][c] = last, cur = suf[cur];
16            if (cur == -1) {
17                suf[last] = 0;
18                continue;
19            }
20            int nx = g[cur][c];
21            if (len[nx] == len[cur] + 1) {
22                suf[last] = nx;
23                continue;
24            }
25            int cl = sz++;
26            g.push_back(g[nx]);
27            suf.push_back(suf[nx]);
28            len.push_back(len[cur] + 1);
29            suf[last] = suf[nx] = cl;
30            while (cur != -1 && g[cur][c] == nx)
31                g[cur][c] = cl, cur = suf[cur];
32        }
33    }
34 };

```

Kasai

```

1 vector<int> get_lcp(const string& s, const vector<int>& suf){
2     int n = (int)suf.size();
3     vector<int> back(n);

```

```

4   for(int i = 0; i < n; i++) back[suf[i]] = i;
5   vector<int> lcp(n - 1);
6   for(int i = 0, k = 0; i < n; i++){
7       int x = back[i]; k = max(0, k - 1);
8       if(x == n - 1){k = 0; continue;}
9       while(s[suf[x] + k] == s[suf[x + 1] + k]) k++;
10      lcp[x] = k;
11  }
12  return lcp;
13  }

```

Z-function

```

1  vector<int> get_z(const string& s){
2      int n = (int)s.length();
3      vector<int> z(n);
4      for(int i = 1, l = 0, r = 0; i < n; i++){
5          if(i < r) z[i] = min(r - i, z[i - l]);
6          while(i + z[i] < n && s[z[i]] == s[i + z[i]]) z[i]++;
7          if(i + z[i] > r) l = i, r = i + z[i];
8      }
9      return z;
10 }

```

prefix-function

```

1  vector<int> get_pi(const string& s){
2      int n = (int)s.length();
3      vector<int> pr(n);
4      for(int i = 1; i < n; i++){
5          int k = pr[i - 1];
6          while(k && s[k] != s[i]) k = pr[k - 1];
7          if(s[k] == s[i]) k++;
8          pr[i] = k;
9      }
10     return pr;
11 }

```

Aho

```

1  const int A = 300; // alphabet size
2  struct Aho {
3      struct Node {

```

```
4     int nxt[A], go[A];
5     int par, pch, link;
6     int good;
7     Node(): par(-1), pch(-1), link(-1), good(-1) {
8         fill(nxt, nxt + A, -1); fill(go, go + A, -1); }
9 };
10 vec< Node > a;
11 Aho() { a.push_back(Node()); }
12 void add_string(const string &s) {
13     int v = 0;
14     for(char c : s) {
15         if(a[v].nxt[c] == -1) {
16             a[v].nxt[c] = (int)a.size();
17             a.push_back(Node());
18             a.back().par = v;
19             a.back().pch = c;
20         }
21         v = a[v].nxt[c];
22     }
23     a[v].good = 1;
24 }
25 int go(int v, int c) {
26     if(a[v].go[c] == -1) {
27         if(a[v].nxt[c] != -1) {
28             a[v].go[c] = a[v].nxt[c];
29         }else {
30             a[v].go[c] = v ? go(get_link(v), c) : 0;
31         }
32     }
33     return a[v].go[c];
34 }
35 int get_link(int v) {
36     if(a[v].link == -1) {
37         if(!v || !a[v].par) a[v].link = 0;
38         else a[v].link = go(get_link(a[v].par), a[v].pch);
39     }
40     return a[v].link;
41 }
42 bool is_good(int v) {
43     if(!v) return false;
44     if(a[v].good == -1) {
45         a[v].good = is_good(get_link(v));
46     }
47     return a[v].good;
48 }
49 bool is_there_substring(const string &s) {
```

```

50     int v = 0;
51     for(char c : s) {
52         v = go(v, c);
53         if(is_good(v)) {
54             return true;
55         }
56     }
57     return false;
58 }
59 };

```

Hungarian

```

1  vector<int> Hungarian(const vector< vector<int> >& a){ // ALARM: INT everywhere
2      int n = (int)a.size();
3      vector<int> row(n), col(n), pair(n, -1), back(n, -1), prev(n, -1);
4      auto get = [&](int i, int j){ return a[i][j] + row[i] + col[j];};
5      for(int v = 0; v < n; v++){
6          vector<int> min_v(n, v), A_plus(n), B_plus(n);
7          A_plus[v] = 1; int jb;
8          while(true){
9              int pos_i = -1, pos_j = -1;
10             for(int j = 0; j < n; j++){
11                 if(!B_plus[j] && (pos_i == -1 ||
12                     get(min_v[j], j) < get(pos_i, pos_j))) {
13                     pos_i = min_v[j], pos_j = j;
14                 }
15             }
16             int weight = get(pos_i, pos_j);
17             for(int i = 0; i < n; i++) if(!A_plus[i]) row[i] += weight;
18             for(int j = 0; j < n; j++) if(!B_plus[j]) col[j] -= weight;
19             B_plus[pos_j] = 1, prev[pos_j] = pos_i;
20             int x = back[pos_j];
21             if(x == -1) { jb = pos_j; break;}
22             A_plus[x] = 1;
23             for(int j = 0; j < n; j++)
24                 if(get(x, j) < get(min_v[j], j))
25                     min_v[j] = x;
26         }
27         while(jb != -1){
28             back[jb] = prev[jb];
29             swap(pair[prev[jb]], jb);
30         }
31     }

```



```

32     return pair;
33 }

```

Hopcroft-Karp

```

1  struct HopcroftKarp {
2      int n, m;
3      vec< vec< int > > g;
4      vec< int > pl, pr, dist;
5      HopcroftKarp(): n(0), m(0) { }
6      HopcroftKarp(int _n, int _m): n(_n), m(_m) { g.resize(n); }
7      void add_edge(int u, int v) { g[u].push_back(v); }
8      bool bfs() {
9          dist.assign(n + 1, inf);
10         queue< int > q;
11         for(int u = 0; u < n; u++) {
12             if(pl[u] < m) continue;
13             dist[u] = 0;
14             q.push(u);
15         }
16         while(!q.empty()) {
17             int u = q.front();
18             q.pop();
19             if(dist[u] >= dist[n]) continue;
20             for(int v : g[u]) {
21                 if(dist[ pr[v] ] > dist[u] + 1) {
22                     dist[ pr[v] ] = dist[u] + 1;
23                     q.push(pr[v]);
24                 }
25             }
26         }
27         return dist[n] < inf;
28     }
29     bool dfs(int v) {
30         if(v == n) return 1;
31         for(int to : g[v]) {
32             if(dist[ pr[to] ] != dist[v] + 1) continue;
33             if(!dfs(pr[to])) continue;
34             pl[v] = to;
35             pr[to] = v;
36             return 1;
37         }
38         return 0;
39     }
40     int find_max_matching() {

```

```

41     pl.resize(n, m);
42     pr.resize(m, n);
43     int result = 0;
44     while(bfs()) {
45         for(int u = 0; u < n; u++) {
46             if(pl[u] < m) continue;
47             result += dfs(u);
48         }
49     }
50     return result;
51 }
52 };

```

CHT

```

1  struct Line {
2      ll k, b;
3      int type;
4      ld x;
5      Line(): k(0), b(0), type(0), x(0) { }
6      Line(ll _k, ll _b, ld _x = 1e18, int _type = 0):
7          k(_k), b(_b), x(_x), type(_type) { }
8      bool operator<(const Line& other) const {
9          if(type + other.type > 0) { return x < other.x;
10             }else { return k < other.k; }
11     }
12     ld intersect(const Line& other) const {
13         return ld(b - other.b) / ld(other.k - k);
14     }
15     ll get_func(ll x0) const {
16         return k * x0 + b;
17     }
18 };
19 struct CHT {
20     set< Line > qs;
21     set< Line > :: iterator fnd, help;
22     bool hasr(const set< Line > :: iterator& it) {
23         return it != qs.end() && next(it) != qs.end(); }
24     bool hasl(const set< Line > :: iterator& it) {
25         return it != qs.begin(); }
26     bool check(const set< Line > :: iterator& it) {
27         if(!hasr(it)) return true;
28         if(!hasl(it)) return true;
29         return it->intersect(*prev(it)) < it->intersect(*next(it)); }
30     void update_intersect(const set< Line > :: iterator& it) {

```

```

31     if(it == qs.end()) return;
32     if(!hasr(it)) return;
33     Line tmp = *it;
34     tmp.x = tmp.intersect(*next(it));
35     qs.insert(qs.erase(it), tmp);
36 }
37 void add_line(Line L) {
38     if(qs.empty()) { qs.insert(L); return; }
39     { fnd = qs.lower_bound(L);
40         if(fnd != qs.end() && fnd->k == L.k) {
41             if(fnd->b >= L.b) return;
42             else qs.erase(fnd);
43         }
44     }
45     fnd = qs.insert(L).first;
46     if(!check(fnd)) { qs.erase(fnd); return; }
47     while(hasr(fnd) && !check(help = next(fnd))) { qs.erase(help); }
48     while(hasl(fnd) && !check(help = prev(fnd))) { qs.erase(help); }
49     if(hasl(fnd)) { update_intersect(prev(fnd)); }
50     update_intersect(fnd);
51 }
52 ll get_max(ld x0) {
53     if(qs.empty()) return -inf64;
54     fnd = qs.lower_bound(Line(0, 0, x0, 1));
55     if(fnd == qs.end()) fnd--;
56     ll res = -inf64; int i = 0;
57     while(i < 2 && fnd != qs.end()) {
58         res = max(res, fnd->get_func(x0));
59         fnd++;
60         i++;
61     }
62     while(i-- > 0) fnd--;
63     while(i < 2) {
64         res = max(res, fnd->get_func(x0));
65         if(hasl(fnd)) {
66             fnd--; i++;
67         }else {
68             break;
69         }
70     }
71     return res;
72 }
73 };

```

FFT with prime mod

```

1  const int mod = 998244353;
2  const int root = 31;
3  const int LOG = 23;
4  const int N = 1e5 + 5;
5  vec< int > G[LOG + 1];
6  vec< int > rev[LOG + 1];
7  inline void _add(int &x, int y);
8  inline int _sum(int a, int b);
9  inline int _sub(int a, int b);
10 inline int _mul(int a, int b);
11 inline int _binpow(int x, int p);
12 inline int _rev(int x);
13 void precalc() {
14     for(int start = root, lvl = LOG; lvl >= 0; lvl--, start = _mul(start, start)) {
15         int tot = 1 << lvl;
16         G[lvl].resize(tot);
17         for(int cur = 1, i = 0; i < tot; i++, cur = _mul(cur, start)) {
18             G[lvl][i] = cur;
19         }
20     }
21     for(int lvl = 1; lvl <= LOG; lvl++) {
22         int tot = 1 << lvl;
23         rev[lvl].resize(tot);
24         for(int i = 1; i < tot; i++) {
25             rev[lvl][i] = ((i & 1) << (lvl - 1)) | (rev[lvl][i >> 1] >> 1);
26         }
27     }
28 }
29 void fft(vec< int > &a, int sz, bool invert) {
30     int n = 1 << sz;
31     for(int j, i = 0; i < n; i++) {
32         if((j = rev[sz][i]) < i) {
33             swap(a[i], a[j]);
34         }
35     }
36     for(int f1, f2, lvl = 0, len = 1; len < n; len <= 1, lvl++) {
37         for(int i = 0; i < n; i += (len << 1)) {
38             for(int j = 0; j < len; j++) {
39                 f1 = a[i + j];
40                 f2 = _mul(a[i + j + len], G[lvl + 1][j]);
41                 a[i + j] = _sum(f1, f2);
42                 a[i + j + len] = _sub(f1, f2);
43             }
44         }
45     }

```

```

46     if(invert) {
47         reverse(a.begin() + 1, a.end());
48         int rn = _rev(n);
49         for(int i = 0; i < n; i++) {
50             a[i] = _mul(a[i], rn);
51         }
52     }
53 }
54 vec< int > multiply(const vec< int > &a, const vec< int > &b) {
55     vec< int > fa(ALL(a));
56     vec< int > fb(ALL(b));
57     int n = (int)a.size();
58     int m = (int)b.size();
59     int maxnm = max(n, m), sz = 0;
60     while((1 << sz) < maxnm) sz++; sz++;
61     fa.resize(1 << sz);
62     fb.resize(1 << sz);
63     fft(fa, sz, false);
64     fft(fb, sz, false);
65     int SZ = 1 << sz;
66     for(int i = 0; i < SZ; i++) { fa[i] = _mul(fa[i], fb[i]); }
67     fft(fa, sz, true);
68     while((int)fa.size() > 1 && !fa.back()) fa.pop_back();
69     return fa;
70 }

```

FFT with ld

```

1  struct Complex {
2      ld rl = 0, im = 0;
3      Complex() = default;
4      Complex(ld x, ld y = 0): rl(x), im(y) { }
5      Complex & operator = (const Complex &o) {
6          if(this != &o) {
7              rl = o.rl;
8              im = o.im;
9          }
10         return *this;
11     }
12     Complex operator + (const Complex &o) const { return {rl + o.rl, im + o.im}; }
13     Complex operator - (const Complex &o) const { return {rl - o.rl, im - o.im}; }
14     Complex operator * (const Complex &o) const {
15         return {rl * o.rl - im * o.im, rl * o.im + im * o.rl}; }
16     Complex & operator *= (const Complex &o) {
17         (*this) = (*this) * o;

```

```

18     return *this;
19 }
20 Complex operator / (const Complex &o) const {
21     ld md = o.rl * o.rl + o.im * o.im;
22     return {(rl * o.rl + im * o.im) / md, (im * o.rl - rl * o.im) / md};
23 }
24 Complex & operator /= (int k) {
25     rl /= k;
26     im /= k;
27     return *this;
28 }
29 ld real() const { return rl; }
30 ld imag() const { return im; }
31 };
32 typedef Complex base;
33 const int LOG = 20;
34 const int N = 1 << LOG;
35 int rev[N];
36 vec< base > PW[LOG + 1];
37 void precalc() {
38     for(int i = 1; i < N; i++) {
39         rev[i] = (rev[i >> 1] >> 1) | ((i & 1) << (LOG - 1)); }
40     for(int lvl = 0; lvl <= LOG; lvl++) {
41         int sz = 1 << lvl;
42         ld alpha = 2 * pi / sz;
43         base root(cos(alpha), sin(alpha));
44         base cur = 1;
45         PW[lvl].resize(sz);
46         for(int j = 0; j < sz; j++) {
47             PW[lvl][j] = cur;
48             cur *= root;
49         }
50     }
51 }
52 void fft(base *a, bool invert = 0) {
53     for(int j, i = 0; i < N; i++) {
54         if((j = rev[i]) > i) swap(a[i], a[j]);
55     }
56     base u, v;
57     for(int lvl = 0; lvl < LOG; lvl++) {
58         int len = 1 << lvl;
59         for(int i = 0; i < N; i += (len << 1)) {
60             for(int j = 0; j < len; j++) {
61                 u = a[i + j];
62                 v = a[i + j + len] *
63                     (invert?PW[lvl+1][j?(len << 1)-j:0]:PW[lvl+1][j]);

```

```

64         a[i + j] = u + v;
65         a[i + j + len] = u - v;
66     }
67 }
68 }
69 if(invert) {
70     for(int i = 0; i < N; i++) {
71         a[i] /= N;
72     }
73 }
74 }

```

Convex-Hull

```

1  struct Point {
2      ll x, y;
3      Point(){}
4      Point(ll x, ll y): x(x), y(y) {}
5      bool operator <(const Point &a) const {
6          return make_pair(x, y) < make_pair(a.x, a.y); }
7      Point operator +(const Point &a) const { return {x + a.x, y + a.y}; }
8      Point operator -(const Point &a) const { return {x - a.x, y - a.y}; }
9      ll cross_product(const Point &a) const { return x * a.y - y * a.x; }
10     ll len2() const { return x * x + y * y; }
11 };
12 vector<Point> convex_hull(vector<Point> v) {
13     Point O = v[0];
14     int pos = 0;
15     for (int i = 0; i < v.size(); i++) {
16         if (v[i] < O)
17             tie(pos, O) = {i, v[i]};
18     }
19     swap(v[0], v[pos]);
20     v = {v.begin() + 1, v.end()};
21     sort(v.begin(), v.end(), [&O](const Point &a, const Point &b){
22         ll prod = (a - O).cross_product(b - O);
23         if (prod)
24             return prod > 0;
25         return (a - O).len2() < (b - O).len2();
26     });
27     vector<Point> ret{O};
28     for (Point &p : v) {
29         while (ret.size() > 1) {
30             Point fr = p - ret[ret.size() - 2],
31                 sc = ret[ret.size() - 1] - ret[ret.size() - 2];

```

```

32         ll prod = sc.cross_product(fr);
33         if (prod > 0)
34             break;
35         ret.pop_back();
36     }
37     ret.push_back(p);
38 }
39 return ret;
40 };

```

Palindrome-Tree

```

1 struct pal_tree {
2     int sz;
3     vector<unordered_map<int,int>> g;
4     vector<int> len, suf, cn, base;
5     vector<int> fin;
6     int cur, v;
7     pal_tree(): g(2),sz(2),len({-1, 0}),cn({0, 0}),suf({0, 0}),cur(0),v(1){}
8     void add_string (const string &s) {
9         for (int i = 0; i < s.length(); i++) {
10             char c = s[i] - 'a';
11             while (i - len[v] - 1 < 0 || s[i - len[v] - 1] - 'a' != c)
12                 v = suf[v];
13             if (!g[v].count(c)) {
14                 g.emplace_back();
15                 int t = len[v];
16                 len.push_back(t + 2);
17                 int u = v;
18                 do {
19                     u = suf[u];
20                 } while (u && s[i - len[u] - 1] - 'a' != c);
21                 suf.push_back(!g[u].count(c) ? 1 : g[u][c]);
22                 t = cn[suf.back()];
23                 cn.push_back(t + 1);
24                 g[v][c] = sz++;
25                 cur++;
26             }
27             v = g[v][c];
28             fin.push_back(cn[v]);
29         }
30         return;
31     }
32 };

```


STL TREE, HASH_MAP

```

1  #include <ext/pb_ds/assoc_container.hpp>
2  #include <ext/pb_ds/tree_policy.hpp>
3  using namespace __gnu_pbds;
4  using namespace std;
5  typedef
6      tree<
7          pair< int, int >,
8          null_type,
9          less< pair< int, int > >,
10         rb_tree_tag,
11         tree_order_statistics_node_update
12     > stat_set;
13 // ...
14 ordered_set X;
15     X.insert(1);
16     X.insert(2);
17     X.insert(4);
18     X.insert(8);
19     X.insert(16);
20     cout<<*X.find_by_order(1)<<endl; // 2
21     cout<<*X.find_by_order(2)<<endl; // 4
22     cout<<*X.find_by_order(4)<<endl; // 16
23     cout<<(end(X)==X.find_by_order(6))<<endl; // true
24     cout<<X.order_of_key(-5)<<endl; // 0
25     cout<<X.order_of_key(1)<<endl; // 0
26     cout<<X.order_of_key(3)<<endl; // 2
27     cout<<X.order_of_key(4)<<endl; // 2
28     cout<<X.order_of_key(400)<<endl; // 5
29 // ...
30 // pbds
31 #include <ext/pb_ds/assoc_container.hpp>
32 #include <ext/pb_ds/detail/standard_policies.hpp>
33 using namespace __gnu_pbds;
34 gp_hash_table<int, int> table; // the usage is the same as the unordered_map

```

Dinic

```

1  struct Dinic {
2      struct Edge { int fr, to, cp, id, fl; };
3      int n, S, T;
4      vec< Edge > es;
5      vec< vec< int > > g;

```

```

6   vec< int > dist, res, ptr;
7   Dinic(int _n, int _S, int _T):
8       n(_n), S(_S), T(_T) { g.resize(n); }
9   void add_edge(int fr, int to, int cp, int id) {
10       g[fr].push_back((int)es.size());
11       es.push_back({fr, to, cp, id, 0});
12       g[to].push_back((int)es.size());
13       es.push_back({to, fr, 0, -1, 0});
14   }
15   bool bfs(int K) {
16       dist.assign(n, inf);
17       dist[S] = 0;
18       queue< int > q;
19       q.push(S);
20       while(!q.empty()) {
21           int v = q.front();
22           q.pop();
23           for(int ps : g[v]) {
24               Edge &e = es[ps];
25               if(e.fl + K > e.cp) continue;
26               if(dist[e.to] > dist[e.fr] + 1) {
27                   dist[e.to] = dist[e.fr] + 1;
28                   q.push(e.to);
29               }
30           }
31       }
32       return dist[T] < inf;
33   }
34   int dfs(int v, int _push = INT_MAX) {
35       if(v == T || !_push) return _push;
36       for(int &iter = ptr[v]; iter < (int)g[v].size(); iter++) {
37           int ps = g[v][ ptr[v] ];
38           Edge &e = es[ps];
39           if(dist[e.to] != dist[e.fr] + 1) continue;
40           int tmp = dfs(e.to, min(_push, e.cp - e.fl));
41           if(tmp) {
42               e.fl += tmp;
43               es[ps ^ 1].fl -= tmp;
44               return tmp;
45           }
46       }
47       return 0;
48   }
49   ll find_max_flow() {
50       ptr.resize(n);
51       ll max_flow = 0, add_flow;

```

```

52     for(int K = 1 << 30; K > 0; K >= 1) {
53         while(bfs(K)) {
54             ptr.assign(n, 0);
55             while((add_flow = dfs(S))) {
56                 max_flow += add_flow;
57             }
58         }
59     }
60     return max_flow;
61 }
62 void assign_result() {
63     res.resize(es.size());
64     for(Edge e : es) if(e.id != -1) res[e.id] = e.fl; }
65 int get_flow(int id) { return res[id]; }
66 bool go(int v, vec< int > &F, vec< int > &path) {
67     if(v == T) return 1;
68     for(int ps : g[v]) {
69         if(F[ps] <= 0) continue;
70         if(go(es[ps].to, F, path)) {
71             path.push_back(ps);
72             return 1;
73         }
74     }
75     return 0;
76 }
77 vec< pair< int, vec< int > > > decomposition() {
78     find_max_flow();
79     vec< int > F((int)es.size()), path, add;
80     vec< pair< int, vec< int > > > dcmp;
81     for(int i = 0; i < (int)es.size(); i++) F[i] = es[i].fl;
82     while(go(S, F, path)) {
83         int mn = INT_MAX;
84         for(int ps : path) mn = min(mn, F[ps]);
85         for(int ps : path) F[ps] -= mn;
86         for(int ps : path) add.push_back(es[ps].id);
87         reverse(ALL(add));
88         dcmp.push_back({mn, add});
89         add.clear();
90         path.clear();
91     }
92     return dcmp;
93 }
94 };

```

MCMF

```

1 struct MCMF {
2     struct Edge { int fr, to, cp, fl, cs, id; };
3     int n, S, T;
4     vec< Edge > es;
5     vec< vec< int > > g;
6     vec< ll > dist, phi;
7     vec< int > from;
8     MCMF(int _n, int _S, int _T): n(_n), S(_S), T(_T)
9     { g.resize(n); }
10    void add_edge(int fr, int to, int cp, int cs, int id) {
11        g[fr].push_back((int)es.size());
12        es.push_back({fr, to, cp, 0, cs, id});
13        g[to].push_back((int)es.size());
14        es.push_back({to, fr, 0, 0, -cs, -1});
15    }
16    void init_phi() {
17        dist.assign(n, LLONG_MAX);
18        dist[S] = 0;
19        for(int any, iter = 0; iter < n - 1; iter++) { // Ford Bellman
20            any = 0;
21            for(Edge e : es) {
22                if(e.fl == e.cp) continue;
23                if(dist[e.to] - dist[e.fr] > e.cs) {
24                    dist[e.to] = dist[e.fr] + e.cs;
25                    any = 1;
26                }
27            }
28            if(!any) break;
29        }
30        phi = dist;
31    }
32    bool Dijkstra() {
33        dist.assign(n, LLONG_MAX);
34        from.assign(n, -1);
35        dist[S] = 0;
36        priority_queue< pair< ll, int >, vec< pair< ll, int > >,
37            greater< pair< ll, int > > > pq;
38        pq.push({dist[S], S});
39        while(!pq.empty()) {
40            int v;
41            ll di;
42            tie(di, v) = pq.top();
43            pq.pop();
44            if(di != dist[v]) continue;
45            for(int ps : g[v]) {

```

```

46         Edge &e = es[ps];
47         if(e.fl == e.cp) continue;
48         if(dist[e.to] - dist[e.fr] > e.cs + phi[e.fr] - phi[e.to]) {
49             dist[e.to] = dist[e.fr] + e.cs + phi[e.fr] - phi[e.to];
50             from[e.to] = ps;
51             pq.push({dist[e.to], e.to});
52         }
53     }
54 }
55 for(int v = 0; v < n; v++) {
56     phi[v] += dist[v];
57 }
58 return dist[T] < LLONG_MAX;
59 }
60 pll find_mcmf() {
61     init_phi();
62     ll flow = 0, cost = 0;
63     while(Dijkstra()) {
64         int mn = INT_MAX;
65         for(int v = T; v != S; v = es[ from[v] ].fr) {
66             mn = min(mn, es[from[v]].cp - es[from[v]].fl);
67         }
68         flow += mn;
69         for(int v = T; v != S; v = es[ from[v] ].fr) {
70             es[ from[v] ].fl += mn;
71             es[ from[v] ^ 1 ].fl -= mn;
72         }
73     }
74     for(Edge &e : es) {
75         if(e.fl >= 0)
76             cost += 1ll * e.fl * e.cs;
77     }
78     return make_pair(flow, cost);
79 }
80 bool go(int v, vec< int > &F, vec< int > &path, vec< int > &used) {
81     if(used[v]) return 0;
82     used[v] = 1;
83     if(v == T) return 1;
84     for(int ps : g[v]) {
85         if(F[ps] <= 0) continue;
86         if(go(es[ps].to, F, path, used)) {
87             path.push_back(ps);
88             return 1;
89         }
90     }
91     return 0;

```

```

92     }
93     vec< pair< int, vec< int > > > decomposition(ll &_flow, ll &_cost) {
94         tie(_flow, _cost) = find_mcmf();
95         vec< int > F((int)es.size()), path, add, used(n);
96         vec< pair< int, vec< int > > > dcmp;
97         for(int i = 0; i < (int)es.size(); i++) F[i] = es[i].fl;
98         while(go(S, F, path, used)) {
99             used.assign(n, 0);
100             int mn = INT_MAX;
101             for(int ps : path) mn = min(mn, F[ps]);
102             for(int ps : path) F[ps] -= mn;
103             for(int ps : path) add.push_back(es[ps].id);
104             reverse(ALL(add));
105             dcmp.push_back({mn, add});
106             add.clear();
107             path.clear();
108         }
109         return dcmp;
110     }
111 };

```

Pollard

```

1  namespace FACTORIZE {
2      const ll MAXX = 1000;
3      const int FERMA_ITER = 30;
4      // const int POLLARD_PO_ITER = 10000;
5      int POLLARD_PO_ITER;
6      inline ll sqr(ll n) { return n * n; }
7      ll check_small(ll n) {
8          for(ll x = 1; sqr(x) <= n && x <= MAXX; x++) {
9              if(x > 1 && n % x == 0) {
10                 return x;
11             } else if(sqr(x + 1) > n) {
12                 return -1;
13             }
14         }
15         return -1;
16     }
17     ll check_square(ll n) {
18         ll bl = 0;
19         ll br = 3e9+1;
20         ll bm;
21         while(br - bl > 1) {
22             bm = (bl + br) / 2;

```

```

23         if(sqr(bm) <= n) {
24             bl = bm;
25         }else {
26             br = bm;
27         }
28     }
29     if(sqr(bl) == n && bl > 1) {
30         return bl;
31     }else {
32         return -1;
33     }
34 }
35 inline ll _mul(ll a, ll b, ll m) {
36     static __int128 xa = 1;
37     static __int128 xb = 1;
38     static __int128 xm = 1;
39     xa = a;
40     xb = b;
41     xm = m;
42     return ll(xa * xb % xm);
43 }
44 /*
45 ll _mul(ll x, ll y, ll mod) {
46     ll q = ld(x) * ld(y) / ld(mod);
47     ll r = x * y - q * mod;
48     return (r % mod + mod) % mod;
49 }*/
50 inline ll _binpow(ll x, ll p, ll m) {
51     static ll res = 1;
52     static ll tmp = 1;
53     res = 1;
54     tmp = x;
55     while(p > 0) {
56         if(p & 1ll) {
57             res = _mul(res, tmp, m);
58         }
59         tmp = _mul(tmp, tmp, m);
60         p >>= 1;
61     }
62     return res;
63 }
64 mt19937_64 next_rand(179);
65 ll gcd(ll x, ll y) { return !x ? y : gcd(y % x, x); }
66 bool is_prime(ll n) {
67     if(n <= 1) return false;
68     if(n == 2) return true;

```

```
69     ll a, g;
70     for(int iter = 0; iter < FERMA_ITER; iter++) {
71         a = next_rand() % (n - 2);
72         if(a < 0) a += n - 2;
73         a += 2;
74         assert(1 < a && a < n);
75         g = gcd(a, n);
76         if(g != 1) { return false; }
77         if(_binpow(a, n - 1, n) != 1) { return false; }
78     }
79     return true;
80 }
81 inline ll _func(ll x, ll n) {
82     static ll result = 1;
83     result = _mul(x, x, n);
84     return result + 1 < n ? result + 1 : 0;
85 }
86 ll pollard_po(ll n) {
87     POLLARD_PO_ITER = 5 + 3 * pow(n, 0.25);
88     ll a, b, x, g;
89     while(1) {
90         a = next_rand() % n;
91         if(a < 0) a += n;
92         b = next_rand() % n;
93         if(b < 0) b += n;
94         for(int iter = 0; iter < POLLARD_PO_ITER; iter++) {
95             x = a >= b ? a - b : b - a;
96             g = gcd(x, n);
97             if(1 < g && g < n) {
98                 return g;
99             }
100             a = _func(a, n);
101             b = _func(_func(b, n), n);
102         }
103     }
104 }
105 ll get_div(ll n) {
106     ll res;
107     res = check_small(n);
108     if(res != -1) { return res; }
109     res = check_square(n);
110     if(res != -1) { return res; }
111     if(is_prime(n)) { return n; }
112     return pollard_po(n);
113 }
```


114

}
