Prime numbers

10^1	$+1$	$+3$	$+7$	$+9$	$+13$	$+19$	$+21$	$+27$	-3	-5	-7	-8
10^2	$+1$	$+3$	$+7$	$+9$	$+13$	$+27$	$+31$	$+37$	-3	-11	-17	-21
10^3	$+9$	$+13$	$+19$	$+21$	$+31$	$+33$	$+39$	$+49$	-3	-9	-17	-23
10^4	$+7$	$+9$	$+37$	$+39$	$+61$	$+67$	$+69$	$+79$	-27	-33	-51	-59
10^5	$+3$	$+19$	$+43$	$+49$	$+57$	$+69$	$+103$	$+109$	-9	-11	-29	-39
10^6	$+3$	$+33$	$+37$	$+39$	$+81$	$+99$	$+117$	$+121$	-17	-21	-39	-41
10^7	$+19$	$+79$	$+103$	$+121$	$+139$	$+141$	$+169$	$+189$	-9	-27	-29	-57
10^8	$+7$	$+37$	$+39$	$+49$	$+73$	$+81$	$+123$	$+127$	-11	-29	-41	-59
10^9	$+7$	$+9$	$+21$	$+33$	$+87$	$+93$	$+97$	$+103$	-63	-71	-107	-117
10^{10}	$+19$	$+33$	$+61$	$+69$	$+97$	$+103$	$+121$	$+141$	-33	-57	-71	-119
10^{11}	$+3$	$+19$	$+57$	$+63$	$+69$	$+73$	$+91$	$+103$	-23	-53	-57	-93
10^{12}	$+39$	$+61$	$+63$	$+91$	$+121$	$+163$	$+169$	$+177$	-11	-39	-41	-63
10^{13}	$+37$	$+51$	$+99$	$+129$	$+183$	$+259$	$+267$	$+273$	-29	-137	-201	-237
10^{14}	$+31$	$+67$	$+97$	$+99$	$+133$	$+139$	$+169$	$+183$	-27	-29	-41	-69
10^{15}	$+37$	$+91$	$+159$	$+187$	$+223$	$+241$	$+249$	$+259$	-11	-53	-117	-123
10^{16}	$+61$	$+69$	$+79$	$+99$	$+453$	$+481$	$+597$	$+613$	-63	-83	-113	-149
10^{17}	$+3$	$+13$	$+19$	$+21$	$+49$	$+81$	$+99$	$+141$	-3	-23	-39	-57
10^{18}	$+3$	$+9$	$+31$	$+79$	$+177$	$+183$	$+201$	$+283$	-11	-33	-123	-137

Primitive Roots

mod	$12 \cdot 2^{10}+1$	$13 \cdot 2^{10}+1$	$15 \cdot 2^{10}+1$	$57 \cdot 2^{10}+1$	$58 \cdot 2^{10}+1$	$60 \cdot 2^{10}+1$	$148 \cdot 2^{10}+1$
root	49	7	84	29	9	21	38
mod	$6 \cdot 2^{11}+1$	$9 \cdot 2^{11}+1$	$20 \cdot 2^{11}+1$	$56 \cdot 2^{11}+1$	$65 \cdot 2^{11}+1$	$140 \cdot 2^{11}+1$	$150 \cdot 2^{11}+1$
root	7	19	32	16	39	106	91
mod	$3 \cdot 2^{12}+1$	$10 \cdot 2^{12}+1$	$15 \cdot 2^{12}+1$	$66 \cdot 2^{12}+1$	$70 \cdot 2^{12}+1$	$75 \cdot 2^{12}+1$	$127 \cdot 2^{12}+1$
root	41	28	19	114	19	41	71
mod	$136 \cdot 2^{12}+1$	$141 \cdot 2^{12}+1$	$5 \cdot 2^{13}+1$	$8 \cdot 2^{13}+1$	$14 \cdot 2^{13}+1$	$51 \cdot 2^{13}+1$	$78 \cdot 2^{13}+1$
root	66	114	12	13	2	67	87
mod	$90 \cdot 2^{13}+1$	$113 \cdot 2^{13}+1$	$4 \cdot 2^{14}+1$	$7 \cdot 2^{14}+1$	$9 \cdot 2^{14}+1$	$63 \cdot 2^{14}+1$	$69 \cdot 2^{14}+1$
root	96	63	15	15	22	94	86
mod	$73 \cdot 2^{14}+1$	$139 \cdot 2^{14}+1$	$2 \cdot 2^{15}+1$	$5 \cdot 2^{15}+1$	$17 \cdot 2^{15}+1$	$81 \cdot 2^{15}+1$	$110 \cdot 2^{15}+1$
root	31	20	9	7	19	89	117
mod	$114 \cdot 2^{15}+1$	$135 \cdot 2^{15}+1$	$1 \cdot 2^{16}+1$	$12 \cdot 2^{16}+1$	$18 \cdot 2^{16}+1$	$55 \cdot 2^{16}+1$	$88 \cdot 2^{16}+1$
root	27	126	3	3	14	30	10
mod	$102 \cdot 2^{16}+1$	$112 \cdot 2^{16}+1$	$117 \cdot 2^{16}+1$	$6 \cdot 2^{17}+1$	$9 \cdot 2^{17}+1$	$21 \cdot 2^{17}+1$	$51 \cdot 2^{17}+1$
root	51	83	15	8	74	83	43
mod	$53 \cdot 2^{17}+1$	$63 \cdot 2^{17}+1$	$104 \cdot 2^{17}+1$	$108 \cdot 2^{17}+1$	$123 \cdot 2^{17}+1$	$3 \cdot 2^{18}+1$	$22 \cdot 2^{18}+1$
root	47	10	13	54	26	5	74
mod	$28 \cdot 2^{18}+1$	$52 \cdot 2^{18}+1$	$54 \cdot 2^{18}+1$	$63 \cdot 2^{18}+1$	$108 \cdot 2^{18}+1$	$127 \cdot 2^{18}+1$	$147 \cdot 2^{18}+1$
root	79	4	25	70	108	99	34
mod	$11 \cdot 2^{19}+1$	$14 \cdot 2^{19}+1$	$26 \cdot 2^{19}+1$	$54 \cdot 2^{19}+1$	$57 \cdot 2^{19}+1$	$71 \cdot 2^{19}+1$	$134 \cdot 2^{19}+1$
root	12	25	2	106	20	86	49
mod	$7 \cdot 2^{20}+1$	$13 \cdot 2^{20}+1$	$22 \cdot 2^{20}+1$	$66 \cdot 2^{20}+1$	$67 \cdot 2^{20}+1$	$106 \cdot 2^{20}+1$	$115 \cdot 2^{20}+1$
root	5	3	50	54	7	85	138
mod	$148 \cdot 2^{20}+1$	$11 \cdot 2^{21}+1$	$33 \cdot 2^{21}+1$	$39 \cdot 2^{21}+1$	$53 \cdot 2^{21}+1$	$54 \cdot 2^{21}+1$	$63 \cdot 2^{21}+1$
root	81	38	45	94	54	134	46
mod	$110 \cdot 2^{21}+1$	$119 \cdot 2^{21}+1$	$123 \cdot 2^{21}+1$	$25 \cdot 2^{22}+1$	$27 \cdot 2^{22}+1$	$33 \cdot 2^{22}+1$	$55 \cdot 2^{22}+1$
root	68	135	95	21	66	30	63
mod	$90 \cdot 2^{22}+1$	$99 \cdot 2^{22}+1$	$20 \cdot 2^{23}+1$	$56 \cdot 2^{23}+1$	$77 \cdot 2^{23}+1$	$107 \cdot 2^{23}+1$	$119 \cdot 2^{23}+1$
root	139	65	4	53	19	45	31
mod	$132 \cdot 2^{23}+1$	$10 \cdot 2^{24}+1$	$28 \cdot 2^{24}+1$	$66 \cdot 2^{24}+1$	$73 \cdot 2^{24}+1$	$108 \cdot 2^{24}+1$	$120 \cdot 2^{24}+1$
root	64	2	40	8	149	126	21
mod	$148 \cdot 2^{24}+1$						
root	25						

Misc

Gomory-Hu tree (Gusfield's algorithm): label nodes from 0 to $(|V|-1)$ and set $p_i = 0 \forall i > 0$. $\forall i > 0$: find min-cut (S, T) between i and p_i, where $i \in S, p_i \in T$; for each node j, s.t. $i < j, j \in S, p_j = p_i$ set $p_j = i$

$\boxed{d_i = v_i - \sum\limits_{j<i} \frac{\langle v_i, d_j \rangle}{\langle d_j, d_j \rangle} d_j}$ $\boxed{\sum\limits_{k=1}^{n} \mu(k) \lfloor \frac{n}{k} \rfloor = 1}$ $\boxed{g(n) = \sum\limits_{d|n} f(d) \Leftrightarrow f(n) = \sum\limits_{d|n} \mu(d) g(n/d)}$

$\boxed{\mu_A^*(a,b) = \begin{cases} 1, & a = b \\ -\sum\limits_{a \preccurlyeq z \prec b} \mu_A^*(a,z), & a \prec b \\ 0, & b \prec a \end{cases}}$ $\boxed{\sum\limits_{d|n} \varphi(d) = n}$ $\boxed{\varphi(n) = \sum\limits_{d|n} d \cdot \mu(\frac{n}{d})}$

$\boxed{F(N) = \sum\limits_{n=1}^{N} \varphi(n) \Rightarrow F(N) = \frac{N(N+1)}{2} - \sum\limits_{k=2}^{N} F(\lfloor \frac{N}{k} \rfloor)}$

$\boxed{Gx = \{y \in X | \exists a \in G \colon a \star x = y\}}$ $\boxed{G_x = \{a \in G \colon a \star x = x\}}$ $\boxed{|G| = |Gx| \cdot |G_x|}$ $\boxed{X^a = \{x \in X \colon a \star x = x\}}$

$\boxed{|X/G| = \frac{1}{|G|} \sum\limits_{a \in G} |X^a|}$ $\boxed{\prod\limits_{k=1}^{\infty} (1 - x^k) = \sum\limits_{q=-\infty}^{+\infty} (-1)^q x^{\frac{3q^2+q}{2}}}$

$\boxed{\prod\limits_{k=1}^{\infty} \frac{1}{1-x^k} = \sum\limits_{n=0}^{+\infty} p(n) x^n \Rightarrow p(0) = 1, p(n) = \sum\limits_{q=1}^{+\infty} (-1)^{q+1} \left[p\left(n - \frac{3q^2-q}{2}\right) + p\left(n - \frac{3q^2+q}{2}\right) \right]}$

$\boxed{M_1 = (S, I_1) \cap M_2 = (S, I_2). J.y \to z (J - y + z \in I_1). y \leftarrow z (J - y + z \in I_2)}$

$X_2 = \{z \in S/J \colon J + z \in I_1\}. X_2 = \{z \in S/J \colon J + z \in I_2\}$. Находим **кратчайший** путь из X_1 в X_2. Ксорим.

$\boxed{x = \frac{B_1 C_2 - B_2 C_1}{A_1 B_2 - A_2 B_1}, y = \frac{A_2 C_1 - A_1 C_2}{A_1 B_2 - A_2 B_1}}$

Теорема Пика. $S = I + \frac{B}{2} - 1$, где S – площадь, B – # на сторонах, I – # строго внутри

Suffix Tree

```cpp
// Ukkonen's algorithm O(n)
const int A = 27; // Alphabet size
struct SuffixTree {
  struct Node { // [l, r) !!!
    int l, r, link, par;
    int nxt[A];
    Node() : l(-1), r(-1), link(-1), par(-1) {
      fill(nxt, nxt + A, -1);
    }
    Node(int _l, int _r, int _link, int _par)
        : l(_l), r(_r), link(_link), par(_par) {
      fill(nxt, nxt + A, -1);
    }
    int& next(int c) { return nxt[c]; }
    int get_len() const { return r - l; }
  };
  struct State {
    int v, len;
  };
  vec<Node> t;
  State cur_state;
  vec<int> s;
  SuffixTree() : cur_state({0, 0}) {
    t.push_back(Node());
  }
  // v -> v + s[l, r) !!!
  State go(State st, int l, int r) {
    while (l < r) {
      if (st.len == t[st.v].get_len()) {
        State nx = State({t[st.v].next(s[l]),
        ↪ 0});
        if (nx.v == -1) return nx;
        st = nx;
        continue;
      }
      if (s[t[st.v].l + st.len] != s[l])
        return State({-1, -1});
      if (r - l < t[st.v].get_len() - st.len)
        return State({st.v, st.len + r - l});
      l += t[st.v].get_len() - st.len;
      st.len = t[st.v].get_len();
    }
    return st;
  }
  int get_vertex(State st) {
    if (t[st.v].get_len() == st.len) return st.v;
    if (st.len == 0) return t[st.v].par;
    Node& v = t[st.v];
    Node& pv = t[v.par];
    Node add(v.l, v.l + st.len, -1, v.par);
    // nxt
    pv.next(s[v.l]) = (int)t.size();
    add.next(s[v.l + st.len]) = st.v;
    // par
    v.par = (int)t.size();
    // [l, r)
    v.l += st.len;
    t.push_back(add); // !!!
    return (int)t.size() - 1;
  }
  int get_link(int v) {
    if (t[v].link != -1) return t[v].link;
    if (t[v].par == -1) return 0;
    int to = get_link(t[v].par);
    to = get_vertex(
      go(State({to, t[to].get_len()}),
         t[v].l + (t[v].par == 0), t[v].r));
    return t[v].link = to;
```

```
  }
  void add_symbol(int c) {
    assert(0 <= c && c < A);
    s.push_back(c);
    while (1) {
      State hlp = go(cur_state, (int)s.size() -
      ↪  1,
                     (int)s.size());
      if (hlp.v != -1) {
        cur_state = hlp;
        break;
      }
      int v = get_vertex(cur_state);
      Node add((int)s.size() - 1, +inf, -1, v);
      t.push_back(add);
      t[v].next(c) = (int)t.size() - 1;
      cur_state.v = get_link(v);
      cur_state.len = t[cur_state.v].get_len();
      if (!v) break;
    }
  }
};
```

Suffix Array

```
const int LOG = 21;
struct SuffixArray {
  string s;
  int n;
  vec<int> p;
  vec<int> c[LOG];
  SuffixArray() : n(0) {}
  SuffixArray(string ss) : s(ss) {
    s.push_back(0);
    n = (int)s.size();
    vec<int> pn, cn;
    vec<int> cnt;
    p.resize(n);
    for (int i = 0; i < LOG; i++)
      c[i].resize(n);
    pn.resize(n);
    cn.resize(n);
    cnt.assign(300, 0);
    for (int i = 0; i < n; i++)
      cnt[s[i]]++;
    for (int i = 1; i < (int)cnt.size(); i++)
      cnt[i] += cnt[i - 1];
    for (int i = n - 1; i >= 0; i--)
      p[--cnt[s[i]]] = i;
    for (int i = 1; i < n; i++) {
      c[0][p[i]] = c[0][p[i - 1]];
      if (s[p[i]] != s[p[i - 1]]) c[0][p[i]]++;
    }
    for (int lg = 0, k = 1; k < n;
         k <<= 1, lg++) {
      for (int i = 0; i < n; i++) {
        if ((pn[i] = p[i] - k) < 0) pn[i] += n;
      }
      cnt.assign(n, 0);
      for (int i = 0; i < n; i++)
        cnt[c[lg][pn[i]]]++;
      for (int i = 1; i < (int)cnt.size(); i++)
        cnt[i] += cnt[i - 1];
      for (int i = n - 1; i >= 0; i--)
        p[--cnt[c[lg][pn[i]]]] = pn[i];
      for (int l1, r1, l2, r2, i = 1; i < n;
           i++) {
        cn[p[i]] = cn[p[i - 1]];
        l1 = p[i - 1];
        l2 = p[i];
        if ((r1 = l1 + k) >= n) r1 -= n;
        if ((r2 = l2 + k) >= n) r2 -= n;
        if (c[lg][l1] != c[lg][l2] ||
            c[lg][r1] != c[lg][r2])
          cn[p[i]]++;
      }
      c[lg + 1] = cn;
    }
    p.erase(p.begin(), p.begin() + 1);
    n--;
  }
  int get_lcp(int i, int j) {
    int res = 0;
    for (int lg = LOG - 1; lg >= 0; lg--) {
      if (i + (1 << lg) > n || j + (1 << lg) > n)
        continue;
      if (c[lg][i] == c[lg][j]) {
        i += (1 << lg);
        j += (1 << lg);
        res += (1 << lg);
      }
    }
    return res;
  }
};
```

Suffix Automation

```
const int ALPHSIZE = 26; // alphabet size
struct SuffixAutomaton {
  struct Node {
    int link, len;
    int next[ALPHSIZE];
    Node() {
      link = -1;
      len = 0;
      for (int i(0); i < ALPHSIZE; i++)
        next[i] = -1;
    }
  };
  string s;
  vector<Node> sa;
  int last;
  SuffixAutomaton() {
    sa.emplace_back();
    last = 0;
    sa[0].len = 0;
    sa[0].link = -1;
    for (int i(0); i < ALPHSIZE; i++)
      sa[0].next[i] = -1;
  }
  void add(const int& c) {
    s.push_back(c + 'a');
```

```
    int cur = (int)sa.size();
    sa.emplace_back();
    sa[cur].len = sa[last].len + 1;
    int p;
    for (p = last; p != -1 && sa[p].next[c] ==
    ↪  -1;
         p = sa[p].link) {
      sa[p].next[c] = cur;
    }
    if (p == -1) {
      sa[cur].link = 0;
    } else {
      int q = sa[p].next[c];
      if (sa[p].len + 1 == sa[q].len) {
        sa[cur].link = q;
      } else {
        int clone = (int)sa.size();
        sa.emplace_back();
        sa[clone].len = sa[p].len + 1;
        sa[clone].link = sa[q].link;
        for (int i(0); i < ALPHSIZE; i++)
          sa[clone].next[i] = sa[q].next[i];
        sa[cur].link = sa[q].link = clone;
        for (; p != -1 && sa[p].next[c] == q;
             p = sa[p].link) {
          sa[p].next[c] = clone;
        }
      }
    }
    last = cur;
  }
};
```

LCP

```
vector<int> get_lcp(const string& s,
                    const vector<int>& suf) {
  int n = (int)suf.size();
  vector<int> back(n);
  for (int i = 0; i < n; i++)
    back[suf[i]] = i;
  vector<int> lcp(n - 1);
  for (int i = 0, k = 0; i < n; i++) {
    int x = back[i];
    k = max(0, k - 1);
    if (x == n - 1) {
      k = 0;
      continue;
    }
    while (s[suf[x] + k] == s[suf[x + 1] + k])
      k++;
    lcp[x] = k;
  }
  return lcp;
}
```

Manacker

```
pair<vector<int>, vector<int>>
manacker(const string& s) {
  // -> {d0, d1}. RUN test!
  int n = (int)s.size();
  vector<int> d0(n), d1(n);
  for (int l = 0, r = -1, i = 0; i < n; i++) {
    d1[i] =
      i <= r ? min(r - i, d1[l + r - i]) : 0;
    while (i >= d1[i] && i + d1[i] < n &&
           s[i - d1[i]] == s[i + d1[i]])
      d1[i]++;
    d1[i]--;
    if (i + d1[i] > r)
      l = i - d1[i], r = i + d1[i];
  }
  for (int l = 0, r = -1, i = 0; i < n; i++) {
    d0[i] =
      i < r ? min(r - i, d0[l + r - i - 1]) : 0;
    while (i >= d0[i] && i + d0[i] + 1 < n &&
           s[i - d0[i]] == s[i + d0[i] + 1])
      d0[i]++;
    if (d0[i] > 0 && i + d0[i] > r)
      l = i - d0[i] + 1, r = i + d0[i];
  }
  return {d0, d1};
}
```

Prefix Function

```
vector<int> get_pi(const string& s) {
  int n = (int)s.length();
  vector<int> pr(n);
  for (int i = 1; i < n; i++) {
    int k = pr[i - 1];
    while (k && s[k] != s[i])
      k = pr[k - 1];
    if (s[k] == s[i]) k++;
    pr[i] = k;
  }
  return pr;
}
```

Z-Function

```
vector<int> get_z(const string& s) {
  int n = (int)s.length();
  vector<int> z(n);
  for (int i = 1, l = 0, r = 0; i < n; i++) {
    if (i < r) z[i] = min(r - i, z[i - l]);
    while (i + z[i] < n && s[z[i]] == s[i +
    ↪  z[i]])
      z[i]++;
    if (i + z[i] > r) l = i, r = i + z[i];
  }
  return z;
}
```

Tandem (Lorentz)

```
struct Tandem {
  int l, r, k;
```

```cpp
  // [l, l + 2 * k) [l + 1, l + 1 + 2 * k) [l
  // + 2, l + 2 + 2 * k), ..., [r, r + 2 * k)
};
vector<int> z_func(const string& s) {
  int n = (int)s.size();
  vector<int> z(n);
  for (int l = 0, r = -1, i = 1; i < n; i++) {
    int k = i > r ? 0 : min(z[i - l], r - i + 1);
    while (i + k < n && s[i + k] == s[k])
      k++;
    z[i] = k;
    if (i + k - 1 > r) {
      r = i + k - 1;
      l = i;
    }
  }
  return z;
}
const int SIZE = (1000006) * 30;
const int MAXL = (1000006) * 4;
Tandem tandems[SIZE], hlp[MAXL];
int tsz;
void rec(const string& s, int L, int R) {
  if (R - L + 1 <= 1) { return; }
  int M = (L + R) / 2;
  rec(s, L, M);
  rec(s, M + 1, R);
  int nu = M - L + 1;
  int nv = R - M;
  string vu =
    s.substr(M + 1, nv) + "#" + s.substr(L, nu);
  string urvr = vu;
  reverse(urvr.begin(), urvr.end());
  vector<int> z1 = z_func(urvr);
  vector<int> z2 = z_func(vu);
  for (int x = L; x <= R; x++) {
    if (x <= M) {
      int k = M + 1 - x;
      int k1 = L < x ? z1[nu - x + L] : 0;
      int k2 = z2[nv + 1 + x - L];
      int lsh = max(0, k - k2);
      int rsh = min(k1, k - 1);
      if (lsh <= rsh) {
        tandems[tsz++] = {x - rsh, x - lsh, k};
      }
    } else {
      int k = x - M;
      int k1 = x < R ? z2[x - M] : 0;
      int k2 = z1[nu + nv - x + M + 1];
      int lsh = max(1, k - k1);
      int rsh = min(k2, k - 1);
      if (lsh <= rsh) {
        tandems[tsz++] = {x - rsh + 1 - k,
                          x - lsh + 1 - k, k};
      }
    }
  }
}
void compress() { // O(n*log(n)*log(n)) can be
                  // replace with count sort
                  // (O(n*log(n))) BE careful
                  ↪ with
                  // ML !!!
  // O(n*log(n)) --> O(n)
  sort(tandems, tandems + tsz,
       [](const Tandem& t1, const Tandem& t2) {
         return t1.k < t2.k ||
                (t1.k == t2.k && t1.l < t2.l);
       });
  int hlp_sz = 0;
  for (int i = 0; i < tsz; i++) {
    int j = i;
    while (j + 1 < tsz &&
           tandems[i].k == tandems[j + 1].k &&
           tandems[j].r + 1 == tandems[j + 1].l)
           ↪ {
      j++;
    }
    hlp[hlp_sz++] = {tandems[i].l, tandems[j].r,
                     tandems[j].k};
    i = j;
  }
  memcpy(tandems, hlp, sizeof(Tandem) * hlp_sz);
  tsz = hlp_sz;
}
void main_lorentz(const string& s) {
  // n = 10^6  time = 1.8 sec MEM = nlog(n) * 12
  // bytes
  int n = (int)s.size();
  tsz = 0;
  rec(s, 0, n - 1);
  compress();
}
```

Aho-Corasick

```cpp
const int A = 300; // alphabet size
struct Aho {
  struct Node {
    int nxt[A], go[A];
    int par, pch, link;
    int good;
    Node()
        : par(-1), pch(-1), link(-1), good(-1) {
      fill(nxt, nxt + A, -1);
      fill(go, go + A, -1);
    }
  };
  vec<Node> a;
  Aho() { a.push_back(Node()); }
  void add_string(const string& s) {
    int v = 0;
    for (char c : s) {
      if (a[v].nxt[c] == -1) {
        a[v].nxt[c] = (int)a.size();
        a.push_back(Node());
        a.back().par = v;
        a.back().pch = c;
      }
      v = a[v].nxt[c];
    }
    a[v].good = 1;
  }
  int go(int v, int c) {
```

```
    if (a[v].go[c] == -1) {
      if (a[v].nxt[c] != -1) {
        a[v].go[c] = a[v].nxt[c];
      } else {
        a[v].go[c] = v ? go(get_link(v), c) : 0;
      }
    }
    return a[v].go[c];
  }
  int get_link(int v) {
    if (a[v].link == -1) {
      if (!v || !a[v].par)
        a[v].link = 0;
      else
        a[v].link =
          go(get_link(a[v].par), a[v].pch);
    }
    return a[v].link;
  }
  bool is_good(int v) {
    if (!v) return false;
    if (a[v].good == -1) {
      a[v].good = is_good(get_link(v));
    }
    return a[v].good;
  }
  bool is_there_substring(const string& s) {
    int v = 0;
    for (char c : s) {
      v = go(v, c);
      if (is_good(v)) { return true; }
    }
    return false;
  }
};
```

Eertree

```
const int N = 2e6 + 5;
struct EerTree {
  char s[N];
  int n;
  int sz;
  int link[N];
  int len[N];
  map<char, int> nxt[N];
  int diff[N];
  int dp[N][2];
  int slink[N];
  int max_suff;
  int ans[N]; // number of partitions into
              // palindromes of even length
  void clr() {
    fill(s, s + N, 0);
    fill(link, link + N, 0);
    fill(len, len + N, 0);
    fill(nxt, nxt + N, map<char, int>());
    fill(diff, diff + N, 0);
    fill((int*)dp, (int*)dp + N * 2, 0);
    fill(slink, slink + N, 0);
    n = 0;
    sz = 0;
    max_suff = 0;
    fill(ans, ans + N, 0);
  }
  EerTree() {
    clr();
    s[0] = '#'; // not in alphabet
    link[0] = 1;
    link[1] = 0;
    len[0] = -1;
    sz = 2;
    ans[0] = 1;
  }
  int get_link(int from) {
    while (s[n] != s[n - len[from] - 1]) {
      from = link[from];
    }
    return from;
  }
  void add_symbol(char c) {
    s[++n] = c;
    max_suff = get_link(max_suff);
    if (!nxt[max_suff].count(c)) {
      {
        int x = get_link(link[max_suff]);
        link[sz] =
          nxt[x].count(c) ? nxt[x][c] : 1;
      }
      len[sz] = len[max_suff] + 2;
      diff[sz] = len[sz] - len[link[sz]];
      slink[sz] = diff[sz] == diff[link[sz]]
                  ? slink[link[sz]]
                  : link[sz];
      nxt[max_suff][c] = sz++;
    }
    max_suff = nxt[max_suff][c];
    for (int x = max_suff; len[x] > 0;
         x = slink[x]) {
      dp[x][0] = dp[x][1] = 0;
      int j = n - (len[slink[x]] + diff[x]);
      _inc(dp[x][j & 1], ans[j]);
      if (diff[x] == diff[link[x]]) {
        _inc(dp[x][0], dp[link[x]][0]);
        _inc(dp[x][1], dp[link[x]][1]);
      }
      _inc(ans[n], dp[x][n & 1]);
    }
  }
};
```

Components of Vertex Duality

```
struct Edge {
  int fr, to, id;
  int get(int v) { return v == fr ? to : fr; }
};
void dfs(const vector<vector<Edge>>& g,
         vector<int>& fup, vector<int>& tin,
         vector<int>& used, int& timer, int v,
         int par = -1) {
  tin[v] = fup[v] = timer++;
  used[v] = 1;
```

```
  for (Edge e : g[v]) {
    int to = e.get(v);
    if (to == par) continue;
    if (used[to]) {
      fup[v] = min(fup[v], tin[to]);
    } else {
      dfs(g, fup, tin, used, timer, to, v);
      fup[v] = min(fup[v], fup[to]);
    }
  }
}
void paintEdges(const vector<vector<Edge>>& g,
                vector<int>& fup,
                vector<int>& tin,
                vector<int>& used,
                vector<int>& colors, int v,
                int curColor, int& maxColor,
                int par = -1) {
  used[v] = 1;
  for (Edge e : g[v]) {
    int to = e.get(v);
    if (to == par) continue;
    if (!used[to]) {
      if (tin[v] <= fup[to]) {
        int tmpColor = maxColor++;
        colors[e.id] = tmpColor;
        paintEdges(g, fup, tin, used, colors, to,
                   tmpColor, maxColor, v);
      } else {
        colors[e.id] = curColor;
        paintEdges(g, fup, tin, used, colors, to,
                   curColor, maxColor, v);
      }
    } else if (tin[to] < tin[v]) {
      colors[e.id] = curColor;
    }
  }
}
vector<vector<Edge>>
get2components(const vector<vector<Edge>>& g,
               int m, const vector<Edge>& es) {
  int n = (int)g.size();
  vector<int> fup(n), tin(n), used(n);
  vector<int> colors(m);
  int timer;
  used.assign(n, 0);
  timer = 0;
  for (int v = 0; v < n; v++) {
    if (used[v]) continue;
    dfs(g, fup, tin, used, timer, v);
  }
  used.assign(n, 0);
  timer = 0;
  for (int v = 0; v < n; v++) {
    if (used[v]) continue;
    paintEdges(g, fup, tin, used, colors, v,
               timer, timer, -1);
  }
  vector<vector<Edge>> res(timer);
  for (int i = 0; i < m; i++) {
    res[colors[i]].push_back(es[i]);
  }
  return res;
}
```

Hungarian Algorihtm

```
vector<int>
Hungarian(const vector<vector<int>>&
            a) { // ALARM: INT everywhere
  int n = (int)a.size();
  vector<int> row(n), col(n), pair(n, -1),
    back(n, -1), prev(n, -1);
  auto get = [&](int i, int j) {
    return a[i][j] + row[i] + col[j];
  };
  for (int v = 0; v < n; v++) {
    vector<int> min_v(n, v), A_plus(n),
    ↪ B_plus(n);
    A_plus[v] = 1;
    int jb;
    while (true) {
      int pos_i = -1, pos_j = -1;
      for (int j = 0; j < n; j++) {
        if (!B_plus[j] && (pos_i == -1 ||
                           get(min_v[j], j) <
                            get(pos_i, pos_j)))
                            ↪ {
          pos_i = min_v[j], pos_j = j;
        }
      }
      int weight = get(pos_i, pos_j);
      for (int i = 0; i < n; i++)
        if (!A_plus[i]) row[i] += weight;
      for (int j = 0; j < n; j++)
        if (!B_plus[j]) col[j] -= weight;
      B_plus[pos_j] = 1, prev[pos_j] = pos_i;
      int x = back[pos_j];
      if (x == -1) {
        jb = pos_j;
        break;
      }
      A_plus[x] = 1;
      for (int j = 0; j < n; j++)
        if (get(x, j) < get(min_v[j], j))
          min_v[j] = x;
    }
    while (jb != -1) {
      back[jb] = prev[jb];
      swap(pair[prev[jb]], jb);
    }
  }
  return pair;
}
```

General Matching

```
struct GeneralMatching { // O(n^3)
  int n = 0, cc = 10;     // [0, n)
  vector<vector<int>> g; // undirected
  vector<int> mt, used, base, p, color;
  queue<int> q;
  GeneralMatching(int nn)
```

```
    : n(nn), mt(n, -1), used(n), base(n), p(n),
      color(n), g(n) {}
  void add_edge(int u, int v) {
    g[u].push_back(v), g[v].push_back(u);
  }
  void add(int v) {
    if (!used[v]) used[v] = 1, q.push(v);
  }
  int get_lca(int u, int v) {
    cc++;
    while (1) {
      u = base[u], color[u] = cc;
      if (mt[u] == -1) break;
      u = p[mt[u]];
    }
    while (1) {
      v = base[v];
      if (color[v] == cc) break;
      v = p[mt[v]];
    }
    return v;
  }
  void mark_path(int v, int child, int b) {
    while (base[v] != b) {
      color[base[v]] = color[base[mt[v]]] = cc;
      p[v] = child, child = mt[v], v = p[child];
    }
  }
  int bfs(int root) {
    add(root);
    while (!q.empty()) {
      int v = q.front();
      q.pop();
      for (int to : g[v]) {
        if (base[v] == base[to] || mt[v] == to)
          continue;
        else if (used[to]) {
          int w = get_lca(v, to);
          cc++, mark_path(v, to, w),
            mark_path(to, v, w);
          for (int i = 0; i < n; i++)
            if (color[base[i]] == cc)
              base[i] = w, add(i);
        } else if (p[to] == -1) {
          p[to] = v;
          if (mt[to] == -1) return to;
          add(mt[to]);
        }
      }
    }
    return -1;
  }
  void xor_path(int v) {
    while (v != -1) {
      int pv = p[v], ppv = mt[pv];
      mt[v] = pv, mt[pv] = v;
      v = ppv;
    }
  }
  bool inc(int root) {
    used.assign(n, 0), p.assign(n, -1),
      iota(base.begin(), base.end(), 0);
    while (!q.empty())
      q.pop();
    int v = bfs(root);
    if (v == -1) return false;
    xor_path(v);
    return true;
  }
  void match() {
    for (int i = 0; i < n; i++)
      if (mt[i] == -1) inc(i);
  }
};
```

Hopcroft-Karp

```
struct HopcroftKarp {
  int n, m;
  vec<vec<int>> g;
  vec<int> pl, pr, dist;
  vec<bool> vis;
  HopcroftCarp() : n(0), m(0) {}
  HopcroftCarp(int _n, int _m) : n(_n), m(_m) {
    g.resize(n);
  }
  void add_edge(int u, int v) {
    g[u].push_back(v);
  }
  bool bfs() {
    dist.assign(n + 1, inf);
    queue<int> q;
    for (int u = 0; u < n; u++) {
      if (pl[u] < m) continue;
      dist[u] = 0;
      q.push(u);
    }
    while (!q.empty()) {
      int u = q.front();
      q.pop();
      if (dist[u] >= dist[n]) continue;
      for (int v : g[u]) {
        if (dist[pr[v]] > dist[u] + 1) {
          dist[pr[v]] = dist[u] + 1;
          q.push(pr[v]);
        }
      }
    }
    return dist[n] < inf;
  }
  bool dfs(int v) {
    if (v == n) return 1;
    vis[v] = true;
    for (int to : g[v]) {
      if (dist[pr[to]] != dist[v] + 1) continue;
      if (vis[pr[to]]) continue;
      if (!dfs(pr[to])) continue;
      pl[v] = to;
      pr[to] = v;
      return 1;
    }
    return 0;
  }
  int find_max_matching() {
```

```
    pl.resize(n, m);
    pr.resize(m, n);
    int result = 0;
    while (bfs()) {
      vis.assign(n + 1, false);
      for (int u = 0; u < n; u++) {
        if (pl[u] < m) continue;
        if (vis[u]) continue;
        result += dfs(u);
      }
    }
    return result;
  }
};
```

Dinic

```
struct Dinic {
  struct Edge {
    int fr, to, cp, id, fl;
  };
  int n, S, T;
  vector<Edge> es;
  vector<vector<int>> g;
  vector<int> dist, res, ptr;
  Dinic(int n_, int S_, int T_)
      : n(n_), S(S_), T(T_) {
    g.resize(n);
  }
  void add_edge(int fr, int to, int cp, int id) {
    g[fr].push_back((int)es.size());
    es.push_back({fr, to, cp, id, 0});
    g[to].push_back((int)es.size());
    es.push_back({to, fr, 0, -1, 0});
  }
  bool bfs(int K) {
    dist.assign(n, inf);
    dist[S] = 0;
    queue<int> q;
    q.push(S);
    while (!q.empty()) {
      int v = q.front();
      q.pop();
      for (int ps : g[v]) {
        Edge& e = es[ps];
        if (e.fl + K > e.cp) continue;
        if (dist[e.to] > dist[e.fr] + 1) {
          dist[e.to] = dist[e.fr] + 1;
          q.push(e.to);
        }
      }
    }
    return dist[T] < inf;
  }
  int dfs(int v, int _push = INT_MAX) {
    if (v == T || !_push) return _push;
    for (int& iter = ptr[v];
         iter < (int)g[v].size(); iter++) {
      int ps = g[v][ptr[v]];
      Edge& e = es[ps];
      if (dist[e.to] != dist[e.fr] + 1) continue;
      int tmp =
          dfs(e.to, min(_push, e.cp - e.fl));
      if (tmp) {
        e.fl += tmp;
        es[ps ^ 1].fl -= tmp;
        return tmp;
      }
    }
    return 0;
  }
  ll find_max_flow() {
    ptr.resize(n);
    ll max_flow = 0, add_flow;
    for (int K = 1 << 30; K > 0; K >>= 1) {
      while (bfs(K)) {
        ptr.assign(n, 0);
        while ((add_flow = dfs(S))) {
          max_flow += add_flow;
        }
      }
    }
    return max_flow;
  }
  void assign_result() {
    res.resize(es.size());
    for (Edge e : es)
      if (e.id != -1) res[e.id] = e.fl;
  }
  int get_flow(int id) { return res[id]; }
  bool go(int v, vector<int>& F,
          vector<int>& path) {
    if (v == T) return 1;
    for (int ps : g[v]) {
      if (F[ps] <= 0) continue;
      if (go(es[ps].to, F, path)) {
        path.push_back(ps);
        return 1;
      }
    }
    return 0;
  }
  vector<pair<int, vector<int>>> decomposition()
  ↪ {
    find_max_flow();
    vector<int> F((int)es.size()), path, add;
    vector<pair<int, vector<int>>> dcmp;
    for (int i = 0; i < (int)es.size(); i++)
      F[i] = es[i].fl;
    while (go(S, F, path)) {
      int mn = INT_MAX;
      for (int ps : path)
        mn = min(mn, F[ps]);
      for (int ps : path)
        F[ps] -= mn;
      for (int ps : path)
        add.push_back(es[ps].id);
      reverse(add.begin(), add.end());
      dcmp.push_back({mn, add});
      add.clear();
      path.clear();
    }
    return dcmp;
  }
```

```
};
```

MCMF

```
struct MCMF {
  struct Edge {
    int fr, to, cp, fl, cs, id;
  };
  int n, S, T;
  vec<Edge> es;
  vec<vec<int>> g;
  vec<ll> dist, phi;
  vec<int> from;
  MCMF(int _n, int _S, int _T)
      : n(_n), S(_S), T(_T) {
    g.resize(n);
  }
  void add_edge(int fr, int to, int cp, int cs,
                int id) {
    g[fr].push_back((int)es.size());
    es.push_back({fr, to, cp, 0, cs, id});
    g[to].push_back((int)es.size());
    es.push_back({to, fr, 0, 0, -cs, -1});
  }
  void init_phi() {
    dist.assign(n, LLONG_MAX);
    dist[S] = 0;
    for (int any, iter = 0; iter < n - 1;
         iter++) { // Ford Bellman
      any = 0;
      for (Edge e : es) {
        if (e.fl == e.cp) continue;
        if (dist[e.to] - dist[e.fr] > e.cs) {
          dist[e.to] = dist[e.fr] + e.cs;
          any = 1;
        }
      }
      if (!any) break;
    }
    phi = dist;
  }
  bool Dijkstra() {
    dist.assign(n, LLONG_MAX);
    from.assign(n, -1);
    dist[S] = 0;
    priority_queue<pair<ll, int>,
                   vec<pair<ll, int>>,
                   greater<pair<ll, int>>>
      pq;
    pq.push({dist[S], S});
    while (!pq.empty()) {
      int v;
      ll di;
      tie(di, v) = pq.top();
      pq.pop();
      if (di != dist[v]) continue;
      for (int ps : g[v]) {
        Edge& e = es[ps];
        if (e.fl == e.cp) continue;
        if (dist[e.to] - dist[e.fr] >
            e.cs + phi[e.fr] - phi[e.to]) {
          dist[e.to] = dist[e.fr] + e.cs +
                       phi[e.fr] - phi[e.to];
          from[e.to] = ps;
          pq.push({dist[e.to], e.to});
        }
      }
    }
    for (int v = 0; v < n; v++) {
      phi[v] += dist[v];
    }
    return dist[T] < LLONG_MAX;
  }
  pll find_mcmf() {
    init_phi();
    ll flow = 0, cost = 0;
    while (Dijkstra()) {
      int mn = INT_MAX;
      for (int v = T; v != S;
           v = es[from[v]].fr) {
        mn = min(mn,
                 es[from[v]].cp -
                 ↪ es[from[v]].fl);
      }
      flow += mn;
      for (int v = T; v != S;
           v = es[from[v]].fr) {
        es[from[v]].fl += mn;
        es[from[v] ^ 1].fl -= mn;
      }
    }
    for (Edge& e : es) {
      if (e.fl >= 0) cost += 1ll * e.fl * e.cs;
    }
    return make_pair(flow, cost);
  }
  bool go(int v, vec<int>& F, vec<int>& path,
          vec<int>& used) {
    if (used[v]) return 0;
    used[v] = 1;
    if (v == T) return 1;
    for (int ps : g[v]) {
      if (F[ps] <= 0) continue;
      if (go(es[ps].to, F, path, used)) {
        path.push_back(ps);
        return 1;
      }
    }
    return 0;
  }
  vec<pair<int, vec<int>>>
  decomposition(ll& _flow, ll& _cost) {
    tie(_flow, _cost) = find_mcmf();
    vec<int> F((int)es.size()), path, add,
      used(n);
    vec<pair<int, vec<int>>> dcmp;
    for (int i = 0; i < (int)es.size(); i++)
      F[i] = es[i].fl;
    while (go(S, F, path, used)) {
      used.assign(n, 0);
      int mn = INT_MAX;
      for (int ps : path)
        mn = min(mn, F[ps]);
      for (int ps : path)
```

```
      F[ps] -= mn;
    for (int ps : path)
      add.push_back(es[ps].id);
    reverse(ALL(add));
    dcmp.push_back({mn, add});
    add.clear();
    path.clear();
  }
  return dcmp;
 }
};
```

Algorithm of Two Chinese

```
struct Edge {
  int fr, to, w, id;
  bool operator<(const Edge& o) const {
    return w < o.w;
  }
};
// find oriented mst (tree)
// there are no edge --> root (root is 0)
// 0 .. n - 1, weights and vertices will be
// changed, but ids are ok
vector<Edge>
work(const vector<vector<Edge>>& graph) {
  int n = (int)graph.size();
  vector<int> color(n), used(n, -1);
  for (int i = 0; i < n; i++)
    color[i] = i;
  vector<Edge> e(n);
  for (int i = 0; i < n; i++) {
    if (graph[i].empty()) {
      e[i] = {-1, -1, -1, -1};
    } else {
      e[i] = *min_element(graph[i].begin(),
                          graph[i].end());
    }
  }
  vector<vector<int>> cycles;
  used[0] = -2;
  for (int s = 0; s < n; s++) {
    if (used[s] != -1) continue;
    int x = s;
    while (used[x] == -1) {
      used[x] = s;
      x = e[x].fr;
    }
    if (used[x] != s) continue;
    vector<int> cycle = {x};
    for (int y = e[x].fr; y != x; y = e[y].fr)
      cycle.push_back(y), color[y] = x;
    cycles.push_back(cycle);
  }
  if (cycles.empty()) return e;
  vector<vector<Edge>> next_graph(n);
  for (int s = 0; s < n; s++) {
    for (const Edge& edge : graph[s]) {
      if (color[edge.fr] != color[s])
        next_graph[color[s]].push_back(
          {color[edge.fr], color[s],
           edge.w - e[s].w, edge.id});
      }
  }
  vector<Edge> tree = work(next_graph);
  for (const auto& cycle : cycles) {
    int cl = color[cycle[0]];
    Edge next_out = tree[cl], out{};
    int from = -1;
    for (int v : cycle) {
      tree[v] = e[v];
      for (const Edge& edge : graph[v])
        if (edge.id == next_out.id)
          from = v, out = edge;
    }
    tree[from] = out;
  }
  return tree;
}
```

Dominator Tree

```
struct Edge {
  int fr = -1;
  int to = -1;
  int id = -1;
};
struct DSU {
  int n = 0; // [0, n)
  vector<int> p, mn;
  DSU() = default;
  DSU(int nn) {
    n = nn;
    p.resize(n);
    mn.resize(n, inf);
    for (int v = 0; v < n; v++)
      p[v] = v;
  }
  void set_value(int v, int x) { mn[v] = x; }
  int find(int v) {
    if (p[v] == v) return v;
    int pv = find(p[v]);
    mn[v] = min(mn[v], mn[p[v]]);
    p[v] = pv;
    return pv;
  }
  void merge(int P, int S) { p[S] = P; }
};
struct DominatorTree {
  int n = 0; // [0, n)
  vector<Edge> edges;
  vector<vector<int>> g, gr;
  vector<int> used, tin, sdom, idom, order,
   ↪  depth;
  DSU dsu;
  vector<vector<int>> cost, parent;
  DominatorTree() = default;
  DominatorTree(int nn) { n = nn; }
  void add_edge(Edge e) { edges.push_back(e); }
  void dfs(int v) {
    used[v] = 1;
    tin[v] = (int)order.size();
    order.push_back(v);
```

```
    for (int eid : g[v]) {
      const auto& e = edges[eid];
      if (!used[e.to]) {
        depth[e.to] = depth[v] + 1;
        parent[0][e.to] = v;
        dfs(e.to);
      }
    }
  }
  void init_binary_jumps() {
    int LOG = 0;
    while ((1 << LOG) < n)
      LOG++;
    cost.resize(LOG, vector<int>(n, inf));
    parent.resize(LOG, vector<int>(n, -1));
  }
  void build_sdom(int s) {
    used.assign(n, 0);
    tin.assign(n, 0);
    depth.assign(n, 0);
    order.clear();
    dfs(s);
    sdom.assign(n, inf);
    idom.assign(n, inf);
    dsu = DSU(n);
    for (int it = (int)order.size() - 1; it >= 0;
         it--) {
      int v = order[it];
      for (int eid : gr[v]) {
        const auto& e = edges[eid];
        if (!used[e.fr]) continue;
        sdom[v] = min(sdom[v], tin[e.fr]);
        if (tin[e.fr] > tin[v]) {
          dsu.find(e.fr);
          sdom[v] = min(sdom[v], dsu.mn[e.fr]);
        }
      }
      dsu.set_value(v, sdom[v]);
      for (int eid : g[v]) {
        const auto& e = edges[eid];
        if (parent[0][e.to] == v) {
          dsu.merge(v, e.to);
        }
      }
    }
  }
  int get_min_on_path(int P, int S) {
    int res = inf;
    for (int j = (int)cost.size() - 1; j >= 0;
         j--) {
      int pS = parent[j][S];
      if (pS == -1 || depth[pS] < depth[P])
        continue;
      res = min(res, cost[j][S]);
      S = pS;
    }
    return res;
  }
  void set_value(int v, int x) {
    cost[0][v] = x;
    for (int j = 1; j < (int)cost.size(); j++) {
      int pv = parent[j - 1][v];
      if (pv == -1) {
        cost[j][v] = cost[j - 1][v];
        parent[j][v] = pv;
      } else {
        cost[j][v] =
          min(cost[j - 1][v], cost[j - 1][pv]);
        parent[j][v] = parent[j - 1][pv];
      }
    }
  }
  void build_idom(int s) {
    for (int v : order) {
      if (v == s) continue;
      idom[v] = min(
        sdom[v], get_min_on_path(order[sdom[v]],
                                 parent[0][v]));
      set_value(v, idom[v]);
    }
  }
  void build(int s) {
    init_binary_jumps();
    g.clear();
    g.resize(n);
    gr.clear();
    gr.resize(n);
    for (int i = 0; i < (int)edges.size(); i++) {
      const auto& e = edges[i];
      g[e.fr].push_back(i);
      gr[e.to].push_back(i);
    }
    build_sdom(s);
    build_idom(s);
  }
};
```

Factorization

```
namespace FACTORIZE {
const ll MAXX = 1000;
const int FERMA_ITER = 30;
// const int POLLARD_PO_ITER = 10000;
int POLLARD_PO_ITER;
inline ll sqr(ll n) { return n * n; }
ll check_small(ll n) {
  for (ll x = 1; sqr(x) <= n && x <= MAXX; x++) {
    if (x > 1 && n % x == 0) {
      return x;
    } else if (sqr(x + 1) > n) {
      return -1;
    }
  }
  return -1;
}
ll check_square(ll n) {
  ll bl = 0;
  ll br = 3e9 + 1;
  ll bm;
  while (br - bl > 1) {
    bm = (bl + br) / 2;
    if (sqr(bm) <= n) {
      bl = bm;
    } else {
```

```
      br = bm;
    }
  }
  if (sqr(bl) == n && bl > 1) {
    return bl;
  } else {
    return -1;
  }
}
inline ll MUL(ll val, ll n, ll mod) {
  long long int q =
    ((double)val * (double)n / (double)mod);
  long long int res = val * n - mod * q;
  res = (res % mod + mod) % mod;
  return res;
}
inline ll _mul(ll a, ll b, ll m) {
  static __int128 xa = 1;
  static __int128 xb = 1;
  static __int128 xm = 1;
  xa = a;
  xb = b;
  xm = m;
  return ll(xa * xb % xm);
}
inline ll _binpow(ll x, ll p, ll m) {
  static ll res = 1;
  static ll tmp = 1;
  res = 1;
  tmp = x;
  while (p > 0) {
    if (p & 1ll) { res = _mul(res, tmp, m); }
    tmp = _mul(tmp, tmp, m);
    p >>= 1;
  }
  return res;
}
mt19937_64 next_rand(42);
ll gcd(ll x, ll y) {
  return !x ? y : gcd(y % x, x);
}
bool is_prime(ll n) {
  if (n <= 1) return false;
  if (n == 2) return true;
  ll a, g;
  for (int iter = 0; iter < FERMA_ITER; iter++) {
    a = next_rand() % (n - 2);
    if (a < 0) a += n - 2;
    a += 2;
    assert(1 < a && a < n);
    g = gcd(a, n);
    if (g != 1) { return false; }
    if (_binpow(a, n - 1, n) != 1) {
      return false;
    }
  }
  return true;
}
inline ll _func(ll x, ll n) {
  static ll result = 1;
  result = _mul(x, x, n);
  return result + 1 < n ? result + 1 : 0;
}
```

```
ll diff(ll x, ll y, ll mod) {
  if (x - y < 0)
    return x - y + mod;
  else
    return x - y + mod;
}
ll pollard_po(ll n) {
  const int POLLARD_PO_ITER =
    5 + 3 * pow(n, 0.25);
  const int MAGIC_LOG = 20;
  while (true) {
    ll x = next_rand() % n;
    for (int i = 0; i < POLLARD_PO_ITER; i++) {
      x = _mul(x, x, n) + 1;
    }
    ll y = _mul(x, x, n) + 1;
    for (int i = 0;
         i < POLLARD_PO_ITER / MAGIC_LOG; i++) {
      ll g = 1;
      for (int j = 0; j < MAGIC_LOG; j++) {
        g = _mul(g, diff(x, y, n), n);
        y = _mul(y, y, n) + 1;
      }
      ll res = __gcd(g, n);
      if (res != 1 && res != n) return res;
    }
  }
}
ll get_div(ll n) {
  ll res;
  res = check_small(n);
  if (res != -1) { return res; }
  res = check_square(n);
  if (res != -1) { return res; }
  if (is_prime(n)) { return n; }
  return pollard_po(n);
}
} // namespace FACTORIZE
```

Square Root in $\mathbb{Z}_p$

```
// Cipolla's algorithm
struct gauss_number {
  ll w, p;
  ll x, y;
  gauss_number() : w(0), p(2), x(0), y(0) {}
  gauss_number(ll _w, ll _p, ll _x, ll _y)
      : w(_w), p(_p), x(_x), y(_y) {
    assert(p > 0);
    w %= p;
    if (w < 0) w += p;
    x %= p;
    if (x < 0) x += p;
    y %= p;
    if (y < 0) y += p;
  }
  gauss_number(const gauss_number& o)
      : w(o.w), p(o.p), x(o.x), y(o.y) {}
  gauss_number
  operator+(const gauss_number& o) const {
    return gauss_number(w, p, _sum(x, o.x, p),
```

```
                        _sum(y, o.y, p));
  }
  gauss_number operator-() const {
    return gauss_number(w, p, !x ? x : p - x,
                        !y ? y : p - y);
  }
  gauss_number
  operator-(const gauss_number& o) const {
    return *this + (-o);
  }
  gauss_number
  operator*(const gauss_number& o) const {
    return gauss_number(
      w, p,
      _sum(_mul(x, o.y, p), _mul(y, o.x, p), p),
      _sum(_mul(y, o.y, p),
           _mul(x, _mul(o.x, w, p), p), p));
  }
};
ll binpow(ll x, ll p, ll m) {
  ll res = 1 % m;
  ll tmp = x % m;
  if (res < 0) res += m;
  if (tmp < 0) tmp += m;
  while (p > 0) {
    if (p & 1) { res = _mul(res, tmp, m); }
    tmp = _mul(tmp, tmp, m);
    p >>= 1;
  }
  return res;
}
gauss_number gauss_pow(gauss_number x, ll p) {
  gauss_number res(x.w, x.p, 0, 1);
  gauss_number tmp(x);
  while (p > 0) {
    if (p & 1) { res = res * tmp; }
    tmp = tmp * tmp;
    p >>= 1;
  }
  return res;
}
ll find_solution(
  ll p,
  ll a) { // x^2 = a (mod p), x = ?, p is prime
  assert(0ll <= a && a < p);
  if (a == 0 || p == 2) return a;
  if (binpow(a, (p - 1) / 2, p) == p - 1)
    return -1ll;
  mt19937_64 rnd(42);
  ll k;
  gauss_number e(a, p, 0, 1);
  while (1) {
    k = rnd() % p;
    if (k < 0) k += p;
    gauss_number y(a, p, 1, k);
    y = gauss_pow(y, (p - 1) / 2);
    y.y = _sub(y.y, 1, p);
    {
      ll re = _mul(y.y, binpow(y.x, p - 2, p),
      ↪  p);
      if (_mul(re, re, p) == a) { return re; }
    }
  }
}
```

Euclid (??)

```
ll rec(ll pos, ll left_len, ll left_cost,
       ll right_len, ll right_cost, ll k) {
  if (!k || !right_len) return pos;
  if (pos >= right_len) {
    ll t = (left_len - pos + right_len - 1) /
           right_len;
    if (t * right_cost + left_cost > k)
      return pos;
    pos += t * right_len - left_len;
    k -= (t * right_cost + left_cost);
  }
  ll nxt_left_len = left_len % right_len;
  ll nxt_left_cost =
    (left_len / right_len) * right_cost +
    left_cost;
  if (nxt_left_len == 0) return pos;
  {
    ll t = pos / nxt_left_len;
    if (t * nxt_left_cost > k)
      return pos -
             nxt_left_len * (k / nxt_left_cost);
    k -= t * nxt_left_cost;
    pos -= t * nxt_left_len;
  }
  return rec(pos, nxt_left_len, nxt_left_cost,
             right_len % nxt_left_len,
             (right_len / nxt_left_len) *
                 nxt_left_cost +
               right_cost,
             k);
}
// finds (nw_st + step * x) % mod --> min, 0 <= x
// <= bound
ll euclid(ll nw_st, ll step, ll mod, ll bound) {
  return rec(nw_st, mod, 0, step, 1, bound);
}
```

Primes on Segment

```
const int X = 1.5e7;
const int MEM_K = 20;
const int MEM_N = 1e5;
int d[X];
vector<int> ps;
int mem[MEM_K][MEM_N];
void precalc() {
  for (int p = 2; p < X; p++) {
    if (!d[p]) ps.push_back(d[p] = p);
    for (int x : ps) {
      if (x > d[p] || x * p >= X) break;
      d[x * p] = x;
    }
    d[p] = d[p - 1] + (d[p] == p);
  }
}
ll rec(ll n, int k) {
```

```
  if (n <= 1) return 0;
  if (k == 0) return n - 1;
  if (ps[k - 1] > n) return 0;
  if (n < X && 1ll * ps[k] * ps[k] > n)
    return d[n] - k;
  if (k < MEM_K && n < MEM_N && mem[k][n])
    return mem[k][n] - 1;
  ll res =
    rec(n, k - 1) - rec(n / ps[k - 1], k - 1) -
     ↪ 1;
  if (k < MEM_K && n < MEM_N) mem[k][n] = res +
   ↪ 1;
  return res;
}
ll get_cnt_primes(
  ll n) { // # primes on [1, n], n <= 10^11, 10
          // queries, ~500ms
  ll m = 1;
  while (m * m < n)
    m++;
  assert(m <= n);
  int k = d[m];
  return k + rec(n, k);
}
```

Pro Euclid

```
// ALL in Z-ring
// T, k > 0 && return (T - k) + (T - 2 * k) + ...
// last, last > 0
ll f(ll T, ll k) {
  ll cnt = T / k;
  return T * cnt - k * cnt * (cnt + 1) / 2;
}
// A, B, C > 0
// |{(x, y): x, y > 0 && Ax + By <= C}|
ll count_triangle(ll A, ll B, ll C) {
  if (A + B > C) return 0;
  if (A > B) swap(A, B);
  ll k = B / A;
  return f(k * C / B, k) +
         count_triangle(A, B - A * k,
                        C - A * (k * C / B));
}
// A, B, C, cx, cy > 0
// |{(x,y) : 1 <= x <= cx && 1 <= y <= cy && Ax +
// By <= C }|
ll count_solutions(ll A, ll B, ll C, ll cx,
                   ll cy) {
  assert(A > 0);
  assert(B > 0);
  if (C <= 0 || cx <= 0 || cy <= 0) return 0;
  if (A * cx + B * cy <= C) return cx * cy;
  if (cx >= C / A && cy >= C / B)
    return count_triangle(A, B, C);
  return count_triangle(A, B, C) -
         count_triangle(A, B, C - B * cy) -
         count_triangle(A, B, C - A * cx);
}
```

FFT with prime mod

```
const int mod = 998244353;
const int root = 31;
const int LOG = 23;
const int N = 1e5 + 5;
vec<int> G[LOG + 1];
vec<int> rev[LOG + 1];
inline void _add(int& x, int y) {
  if ((x += y) >= mod) { x -= mod; }
}
inline int _sum(int a, int b) {
  return a + b < mod ? a + b : a + b - mod;
}
inline int _sub(int a, int b) {
  return a >= b ? a - b : a - b + mod;
}
inline int _mul(int a, int b) {
  return (1ll * a * b) % mod;
}
inline int _binpow(int x, int p) {
  int res = 1;
  int tmp = x;
  while (p > 0) {
    if (p & 1) { res = _mul(res, tmp); }
    tmp = _mul(tmp, tmp);
    p >>= 1;
  }
  return res;
}
inline int _rev(int x) {
  return _binpow(x, mod - 2);
}
void precalc() {
  for (int start = root, lvl = LOG; lvl >= 0;
       lvl--, start = _mul(start, start)) {
    int tot = 1 << lvl;
    G[lvl].resize(tot);
    for (int cur = 1, i = 0; i < tot;
         i++, cur = _mul(cur, start)) {
      G[lvl][i] = cur;
    }
  }
  for (int lvl = 1; lvl <= LOG; lvl++) {
    int tot = 1 << lvl;
    rev[lvl].resize(tot);
    for (int i = 1; i < tot; i++) {
      rev[lvl][i] = ((i & 1) << (lvl - 1)) |
                    (rev[lvl][i >> 1] >> 1);
    }
  }
}
void fft(vec<int>& a, int sz, bool invert) {
  int n = 1 << sz;
  for (int j, i = 0; i < n; i++) {
    if ((j = rev[sz][i]) < i) {
      swap(a[i], a[j]);
    }
  }
  for (int f1, f2, lvl = 0, len = 1; len < n;
       len <<= 1, lvl++) {
    for (int i = 0; i < n; i += (len << 1)) {
      for (int j = 0; j < len; j++) {
```

```
        f1 = a[i + j];
        f2 = _mul(a[i + j + len], G[lvl + 1][j]);
        a[i + j] = _sum(f1, f2);
        a[i + j + len] = _sub(f1, f2);
      }
    }
  }
  if (invert) {
    reverse(a.begin() + 1, a.end());
    int rn = _rev(n);
    for (int i = 0; i < n; i++) {
      a[i] = _mul(a[i], rn);
    }
  }
}
vec<int> multiply(const vec<int>& a,
                  const vec<int>& b) {
  vec<int> fa(ALL(a));
  vec<int> fb(ALL(b));
  int n = (int)a.size();
  int m = (int)b.size();
  int maxnm = max(n, m), sz = 0;
  while ((1 << sz) < maxnm)
    sz++;
  sz++;
  fa.resize(1 << sz);
  fb.resize(1 << sz);
  fft(fa, sz, false);
  fft(fb, sz, false);
  int SZ = 1 << sz;
  for (int i = 0; i < SZ; i++) {
    fa[i] = _mul(fa[i], fb[i]);
  }
  fft(fa, sz, true);
  while ((int)fa.size() > 1 && !fa.back())
    fa.pop_back();
  return fa;
}
```

Polynomial Division

```
// let A = series and A[0] != 0 in Z/pZ, p is
// prime finds (A^{-1}) % x^n
vector<int>
series_inverse(const vector<int>& series, int n,
               ll p) {
  vector<int> current = {_div(1, series[0], p)};
  vector<int> A = {};
  int l = 0;
  while ((int)current.size() < n) {
    while (l < 2 * (int)current.size()) {
      A.push_back(
        l < (int)series.size() ? series[l] : 0);
      l++;
    }
    vector<int> next = multiply(A, current);
    for (int& x : next)
      x = (-x % p + p) % p;
    next[0] = _sum(2 % p, next[0], p);
    next = multiply(next, current);
    for (int& x : next)
      x = (x % p + p) % p;
    next.resize(2 * current.size());
    current = next;
  }
  current.resize(n);
  return current;
}
// calculates a / b
vector<int> division(const vector<int>& a,
                     const vector<int>& b,
                     int p) {
  int n = (int)a.size() - 1; // deg(a)
  int m = (int)b.size() - 1; // deg(b)
  if (n < m) { return {0}; }
  vector<int> ar = a, br = b;
  reverse(ar.begin(), ar.end());
  reverse(br.begin(), br.end());
  ar.resize(n - m + 1);
  br.resize(n - m + 1);
  vector<int> qr =
    series_inverse(br, n - m + 1, p);
  qr = multiply(qr, ar);
  qr.resize(n - m + 1);
  for (int& x : qr)
    x = (x % p + p) % p;
  reverse(qr.begin(), qr.end()); // q = q^r
  return qr;
}
// calculates a - bQ
vector<int> module(const vector<int>& a,
                   const vector<int>& b,
                   const vector<int>& Q, int p) {
  vector<int> r = multiply(b, Q);
  r.resize(b.size());
  for (int i = 0; i < (int)r.size(); i++) {
    int ai = i < (int)a.size() ? a[i] : 0;
    int ri = (r[i] % p + p) % p;
    r[i] = _sub(ai, ri, p);
  }
  return r;
}
```

FFT

```
typedef complex<ld> base;
const int LOG = 20;
const int N = 1 << LOG;
int rev[N];
vec<base> PW[LOG + 1];
void precalc() {
  for (int i = 1; i < N; i++) {
    rev[i] =
      (rev[i >> 1] >> 1) | ((i & 1) << (LOG -
      ↪ 1));
  }
  for (int lvl = 0; lvl <= LOG; lvl++) {
    int sz = 1 << lvl;
    ld alpha = 2 * pi / sz;
    base root(cos(alpha), sin(alpha));
    base cur = 1;
    PW[lvl].resize(sz);
    for (int j = 0; j < sz; j++) {
```

```
      PW[lvl][j] = cur;
      cur *= root;
    }
  }
}
void fft(base* a, bool invert = 0) {
  for (int j, i = 0; i < N; i++) {
    if ((j = rev[i]) > i) swap(a[i], a[j]);
  }
  base u, v;
  for (int lvl = 0; lvl < LOG; lvl++) {
    int len = 1 << lvl;
    for (int i = 0; i < N; i += (len << 1)) {
      for (int j = 0; j < len; j++) {
        u = a[i + j];
        v =
          a[i + j + len] *
          (invert
             ? PW[lvl + 1][j ? (len << 1) - j :
             ↪  0]
             : PW[lvl + 1][j]);
        a[i + j] = u + v;
        a[i + j + len] = u - v;
      }
    }
  }
  if (invert) {
    for (int i = 0; i < N; i++) {
      a[i] /= N;
    }
  }
}
```

Extrapolation

```
int fact[N];
int rfact[N];
void precalc2() {
  fact[0] = 1;
  for (int i = 1; i < N; i++) {
    fact[i] = _mul(fact[i - 1], i);
  }
  rfact[N - 1] = _rev(fact[N - 1]);
  for (int i = N - 2; i >= 0; i--) {
    rfact[i] = _mul(rfact[i + 1], i + 1);
  }
}
int getMulOnSegment(int l, int r) {
  assert(l <= r);
  if (l == 0 && r == 0) return 1;
  if (r <= 0) {
    int res = getMulOnSegment(-r, -l);
    int cnt = r - l + 1;
    if (cnt % 2) {
      res = (-res % mod + mod) % mod;
    }
    return res;
  }
  if (l < 0) {
    int resl = getMulOnSegment(0, -l);
    if (l % 2) {
      resl = (-resl % mod + mod) % mod;
    }
    int resr = getMulOnSegment(0, r);
    return _mul(resl, resr);
  }
  assert(l >= 0);
  int res = fact[r];
  if (l > 0) { res = _mul(res, rfact[l - 1]); }
  return res;
}
vector<int> extrapolate(vector<int> y, int m) {
  vector<int> yy = y;
  int n = (int)y.size() - 1;
  for (int i = 0; i <= n; i++) {
    yy[i] = _mul(
      y[i], _rev(getMulOnSegment(i - n, i - 0)));
  }
  vector<int> ff(n + m + 1);
  for (int i = 1; i <= n + m; i++) {
    ff[i] = _mul(fact[i - 1], rfact[i]);
  }
  vector<int> ss = multiply(yy, ff);
  for (int i = 1; i <= m; i++) {
    int cc = getMulOnSegment(i, n + i);
    int Si = ss[n + i];
    y.push_back(_mul(cc, Si));
  }
  return y;
}
```

Xor FWHT

```
// _sum, _sub, _mul - arithmetic operations
void xor_fwht(vector<int>& a,
              bool inverse = false) {
  for (int x, y, len = 1; len < (int)a.size();
       len <<= 1) {
    for (int i = 0; i < (int)a.size();
         i += len << 1) {
      for (int j = 0; j < len; j++) {
        x = a[i + j], y = a[i + j + len];
        a[i + j] = _sum(x, y);
        a[i + j + len] = _sub(x, y);
      }
    }
  }
  if (inverse) {
    int rn = _binpow((int)a.size(), mod - 2);
    for (int& x : a)
      x = _mul(x, rn);
  }
}
void or_fwht(vector<int>& a,
             bool inverse = false) {
  for (int x, y, len = 1; len < (int)a.size();
       len <<= 1) {
    for (int i = 0; i < (int)a.size();
         i += len << 1) {
      for (int j = 0; j < len; j++) {
        x = a[i + j], y = a[i + j + len];
        a[i + j] = x,
              a[i + j + len] =
```

```
                inverse ? _sub(y, x) : _sum(y,
                 ↪  x);
      }
    }
  }
}
void and_fwht(vector<int>& a,
              bool inverse = false) {
  for (int x, y, len = 1; len < (int)a.size();
       len <<= 1) {
    for (int i = 0; i < (int)a.size();
         i += len << 1) {
      for (int j = 0; j < len; j++) {
        x = a[i + j], y = a[i + j + len];
        a[i + j] =
          inverse ? _sub(x, y) : _sum(x, y),
              a[i + j + len] = y;
      }
    }
  }
}
```

CHT

```
struct Line {
  ll k, b;
  int type;
  ld x;
  Line() : k(0), b(0), type(0), x(0) {}
  Line(ll _k, ll _b, ld _x = 1e18, int _type = 0)
      : k(_k), b(_b), x(_x), type(_type) {}
  bool operator<(const Line& other) const {
    if (type + other.type > 0) {
      return x < other.x;
    } else {
      return k < other.k;
    }
  }
  ld intersect(const Line& other) const {
    return ld(b - other.b) / ld(other.k - k);
  }
  ll get_func(ll x0) const { return k * x0 + b; }
};
struct CHT {
  set<Line> qs;
  set<Line>::iterator fnd, help;
  bool hasr(const set<Line>::iterator& it) {
    return it != qs.end() && next(it) !=
     ↪  qs.end();
  }
  bool hasl(const set<Line>::iterator& it) {
    return it != qs.begin();
  }
  bool check(const set<Line>::iterator& it) {
    if (!hasr(it)) return true;
    if (!hasl(it)) return true;
    return it->intersect(*prev(it)) <
           it->intersect(*next(it));
  }
  void update_intersect(
    const set<Line>::iterator& it) {
    if (it == qs.end()) return;
    if (!hasr(it)) return;
    Line tmp = *it;
    tmp.x = tmp.intersect(*next(it));
    qs.insert(qs.erase(it), tmp);
  }
  void add_line(Line L) {
    if (qs.empty()) {
      qs.insert(L);
      return;
    }
    {
      fnd = qs.lower_bound(L);
      if (fnd != qs.end() && fnd->k == L.k) {
        if (fnd->b >= L.b)
          return;
        else
          qs.erase(fnd);
      }
    }
    fnd = qs.insert(L).first;
    if (!check(fnd)) {
      qs.erase(fnd);
      return;
    }
    while (hasr(fnd) &&
           !check(help = next(fnd))) {
      qs.erase(help);
    }
    while (hasl(fnd) &&
           !check(help = prev(fnd))) {
      qs.erase(help);
    }
    if (hasl(fnd)) {
      update_intersect(prev(fnd));
    }
    update_intersect(fnd);
  }
  ll get_max(ld x0) {
    if (qs.empty()) return -inf64;
    fnd = qs.lower_bound(Line(0, 0, x0, 1));
    if (fnd == qs.end()) fnd--;
    ll res = -inf64;
    int i = 0;
    while (i < 2 && fnd != qs.end()) {
      res = max(res, fnd->get_func(x0));
      fnd++;
      i++;
    }
    while (i-- > 0)
      fnd--;
    while (i < 2) {
      res = max(res, fnd->get_func(x0));
      if (hasl(fnd)) {
        fnd--;
        i++;
      } else {
        break;
      }
    }
    return res;
  }
};
```

Euler Tour Trees

```
class EulerTourTrees {
  /*
  graph - forest
  1 .. n
  get = is connected?
  no memory leaks
  1 <= n, q <= 10^5
  0.7 sec
  */
private:
  struct Node {
    Node* l;
    Node* r;
    Node* p;
    int prior;
    int cnt;
    int rev;
    Node()
        : l(nullptr), r(nullptr), p(nullptr),
          prior(rnd()), cnt(1), rev(0) {}
    ~Node() {
      delete l;
      delete r;
    }
  };
  void do_rev(Node* v) {
    if (v) v->rev ^= 1, swap(v->l, v->r);
  }
  int get_cnt(Node* v) const {
    return v ? v->cnt : 0;
  }
  void update(Node* v) {
    if (!v) return;
    v->cnt = 1 + get_cnt(v->l) + get_cnt(v->r);
    v->p = nullptr;
    if (v->l) v->l->p = v;
    if (v->r) v->r->p = v;
  }
  void push(Node* v) {
    if (!v) return;
    if (v->rev) {
      do_rev(v->l);
      do_rev(v->r);
      v->rev ^= 1;
    }
  }
  void merge(Node*& v, Node* l, Node* r) {
    if (!l || !r) {
      v = l ? l : r;
      return;
    }
    push(l);
    push(r);
    if (l->prior < r->prior) {
      merge(l->r, l->r, r);
      v = l;
    } else {
      merge(r->l, l, r->l);
      v = r;
    }
    update(v);
  }
  void split_by_cnt(Node* v, Node*& l, Node*& r,
                    int x) {
    if (!v) {
      l = r = nullptr;
      return;
    }
    push(v);
    if (get_cnt(v->l) + 1 <= x) {
      split_by_cnt(v->r, v->r, r,
                   x - get_cnt(v->l) - 1);
      l = v;
    } else {
      split_by_cnt(v->l, l, v->l, x);
      r = v;
    }
    update(l);
    update(r);
  }
  void push_path(Node* v) {
    if (!v) return;
    push_path(v->p);
    push(v);
  }
  int get_pos(Node* v) {
    push_path(v);
    int res = 0, ok = 1;
    while (v) {
      if (ok) res += get_cnt(v->l) + 1;
      ok = v->p && v->p->r == v;
      v = v->p;
    }
    return res;
  }
  Node* get_root(Node* v) const {
    while (v && v->p)
      v = v->p;
    return v;
  }
  Node* shift(Node* v) {
    if (!v) return v;
    int pos = get_pos(v);
    Node *nl = nullptr, *nr = nullptr;
    Node* root = get_root(v);
    split_by_cnt(root, nl, nr, pos - 1);
    do_rev(nl);
    do_rev(nr);
    merge(root, nl, nr);
    do_rev(root);
    return root;
  }
public:
  EulerTourTrees() = default;
  EulerTourTrees(int _n) : n(_n) {
    ptr.resize(_n + 1);
    where_edge.resize(_n + 1);
  }
  bool get(int u, int v) const {
    if (u == v) return true;
    Node* ru = get_root(
      ptr[u].empty() ? nullptr :
       ↪ *ptr[u].begin());
    Node* rv = get_root(
```

```
      ptr[v].empty() ? nullptr :
       ↪ *ptr[v].begin());
    return ru && ru == rv;
  }
  void link(int u, int v) {
    Node* ru = shift(
      ptr[u].empty() ? nullptr :
       ↪ *ptr[u].begin());
    Node* rv = shift(
      ptr[v].empty() ? nullptr :
       ↪ *ptr[v].begin());
    Node* uv = new Node();
    Node* vu = new Node();
    ptr[u].insert(uv);
    ptr[v].insert(vu);
    where_edge[u][v] = uv;
    where_edge[v][u] = vu;
    merge(ru, ru, uv);
    merge(ru, ru, rv);
    merge(ru, ru, vu);
  }
  void cut(int u, int v) {
    Node* uv = where_edge[u][v];
    Node* vu = where_edge[v][u];
    ptr[u].erase(uv);
    ptr[v].erase(vu);
    Node* root = shift(uv);
    Node *nl = nullptr, *nm = nullptr,
         *nr = nullptr;
    int pos1 = get_pos(uv);
    int pos2 = get_pos(vu);
    if (pos1 < pos2) {
      split_by_cnt(root, nl, nr, pos2);
      split_by_cnt(nl, nl, vu, pos2 - 1);
      split_by_cnt(nl, nl, nm, pos1);
      split_by_cnt(nl, nl, uv, pos1 - 1);
      merge(nl, nl, nr);
    } else {
      split_by_cnt(root, nl, nr, pos1);
      split_by_cnt(nl, nl, uv, pos1 - 1);
      split_by_cnt(nl, nl, nm, pos2);
      split_by_cnt(nl, nl, vu, pos2 - 1);
      merge(nl, nl, nm);
    }
    delete uv;
    delete vu;
  }
  ~EulerTourTrees() {
    set<Node*> roots;
    for (int i = 1; i <= n; i++) {
      for (Node* v : ptr[i]) {
        roots.insert(get_root(v));
      }
    }
    for (Node* root : roots) {
      delete root;
    }
  }
private:
  int n = 0;
  vec<set<Node*>> ptr;
  vec<unordered_map<int, Node*>>
    where_edge; // ptr to node
};
```

Simplex

```
template <class T>
vector<T> operator+(const vector<T>& a,
                    const vector<T>& b) {
  vector<T> res(a.size());
  for (int i = 0; i < (int)a.size(); i++)
    res[i] = a[i] + b[i];
  return res;
}
template <class T>
vector<T> operator*(const T& coef,
                    const vector<T>& a) {
  vector<T> res(a.size());
  for (int i = 0; i < (int)a.size(); i++)
    res[i] = coef * a[i];
  return res;
}
const ld eps = 1e-9;
struct Simplex {
  // Ax = b, x >= 0, <c, x> -> max
  int m;                // the number of
   ↪ equations
  int n;                // the number of
   ↪ variables
  vector<vector<ld>> A; // (m + 2) x (n + 1)
  // (m + 1)-th row: primary c
  // (m + 2)-th row: seconday c (c')
  // (n + 1)-th col: column of b
  vector<int> basis;
  bool bounded = true;
  Simplex(const vector<vector<ld>>& mat,
          const vector<int>& _basis)
      : A(mat), basis(_basis) {
    m = (int)mat.size() - 2,
    n = (int)mat[0].size() - 1;
  }
  /// make primary c under basis components zero
  void reset_c() {
    for (int i = 0; i < m; i++) {
      int j = basis[i];
      A[m] = A[m] + (-A[m][j]) * A[i];
      A[m + 1] = A[m + 1] + (-A[m + 1][j]) *
       ↪ A[i];
    }
  }
  void pivot(int i, int k) {
    A[k] = (ld(1) / ld(A[k][i])) * A[k];
    for (int j = 0; j < (int)A.size(); j++) {
      if (j == k) continue;
      A[j] = A[j] + (-A[j][i]) * A[k];
    }
    basis[k] = i;
  }
  void run() {
    while (true) {
      int j = 0;
      while (j < n && A[m][j] <= eps)
        j++;
```

```
      if (j == n) break;
      int k = -1;
      for (int i = 0; i < m; i++)
        if (A[i][j] > eps &&
            (k == -1 || (A[i][n] / A[i][j] <
                         A[k][n] / A[k][j])))
          k = i;
      if (k == -1) {
        bounded = false;
        break;
      }
      pivot(j, k);
    }
  }
  vector<ld> get_solution() {
    vector<ld> res(n);
    for (int i = 0; i < m; i++)
      res[basis[i]] = A[i][n];
    return res;
  }
  void reset_column(int j) {
    for (int i = 0; i < (int)A.size(); i++)
      A[i][j] = 0;
  }
  ld get_max_value() { return -A[m][n]; }
  void swap_primary_c() { swap(A[m], A[m + 1]); }
  void flip_task_type() {
    A[m] = ld(-1) * A[m];
    A[m + 1] = ld(-1) * A[m + 1];
  }
};
struct Response {
  bool bounded = true;
  bool exist = true;
  ld value = 0;
  vector<ld> solution = {};
};
// aa * x <= bb, <cc, x> ---> max
Response solve(const vector<vector<ld>>& aa,
               const vector<ld>& bb,
               const vector<ld>& cc) {
  int m = (int)aa.size();
  int n = (int)aa[0].size();
  vector<vector<ld>> a(m,
                        vector<ld>(n + m + 1 +
                        ↪  1));
  for (int i = 0; i < m; i++) {
    for (int j = 0; j < n; j++)
      a[i][j] = aa[i][j];
    a[i][n + i] = +1;
    a[i][n + m] = -1;
    a[i][n + m + 1] = bb[i];
  }
  vector<ld> c(n + m + 1 + 1), c2(n + m + 1 + 1);
  for (int i = 0; i < n; i++)
    c[i] = cc[i];
  c2[n + m] = -1;
  vector<int> basis(m);
  for (int j = 0; j < m; j++)
    basis[j] = n + j;
  a.push_back(c2);
  a.push_back(c);
  Simplex simplex(a, basis);
  simplex.reset_c();
  {
    int k = 0;
    for (int i = 1; i < m; i++)
      if (a[i][n + m + 1] < a[k][n + m + 1])
        k = i;
    if (a[k][n + m + 1] < -eps)
      simplex.pivot(n + m, k);
  }
  simplex.run();
  if (!simplex.bounded ||
      -simplex.get_max_value() > eps) {
    return Response{true, false, 0, {}};
  }
  {
    vector<int> in_basis(n + m + 1, -1);
    for (int i = 0; i < m; i++)
      in_basis[simplex.basis[i]] = i;
    int k = in_basis[n + m];
    if (k != -1) {
      for (int i = 0; i < n + m; i++) {
        if (in_basis[i] != -1) continue;
        if (std::abs(simplex.A[k][i]) <= eps)
          continue;
        simplex.pivot(i, k);
        break;
      }
    }
    simplex.reset_column(n + m);
  }
  simplex.swap_primary_c();
  simplex.run();
  if (!simplex.bounded) {
    return Response{false, true, 0, {}};
  }
  Response response;
  response.value = simplex.get_max_value();
  response.solution = simplex.get_solution();
  response.solution.resize(n);
  return response;
}
```

Fast Allocator

```
#define FAST_ALLOCATOR_MEMORY 4e8
#ifdef FAST_ALLOCATOR_MEMORY
int allocator_pos = 0;
char
↪  allocator_memory[(int)FAST_ALLOCATOR_MEMORY];
inline void* operator new(size_t n) {
  char* res = allocator_memory + allocator_pos;
  allocator_pos += n;
  assert(allocator_pos <=
         (int)FAST_ALLOCATOR_MEMORY);
  return (void*)res;
}
inline void operator delete(void*) noexcept {}
#endif
```

Angle Comparator

```
struct comparator {
  pll center;
  comparator(pll p) : center(p) {}
  bool operator()(const pll& p,
                  const pll& q) const {
    pll start(1, 0);
    if (p == q) return false;
    auto op = vect(center, p);
    auto oq = vect(center, q);
    if (cp(op, oq) == 0 && dp(op, oq) > 0)
      return false;
    ll sop = cp(start, op), soq = cp(start, oq);
    if (sop == 0) {
      if (dp(start, op) > 0) return true;
      return soq < 0;
    }
    if (soq == 0) {
      if (dp(start, oq) > 0) return false;
      return sop > 0;
    }
    if ((sop > 0 && soq > 0) ||
        (sop < 0 && soq < 0)) {
      return cp(op, oq) > 0;
    }
    return sop > 0;
  }
};
```

Minkowsky Polygon Sum

```
vector<pt> minkowski_polygons_sum(vector<pt> a,
                                  vector<pt> b) {
  // a and b have counter-clock wise order
  auto cmp = [](const pt& p1,
                const pt& p2) -> bool {
    return make_pair(p1.x, p1.y) <
           make_pair(p2.x, p2.y);
  };
  rotate(a.begin(),
         min_element(a.begin(), a.end(), cmp),
         a.end());
  rotate(b.begin(),
         min_element(b.begin(), b.end(), cmp),
         b.end());
  pt q = a[0] + b[0];
  int n = (int)a.size();
  int m = (int)b.size();
  vector<pt> result = {q};
  for (int i = 0, j = 0; i < n || j < m;) {
    pt vi, vj;
    if (i < n)
      vi = a[i + 1 < n ? i + 1 : 0] - a[i];
    if (j < m)
      vj = b[j + 1 < m ? j + 1 : 0] - b[j];
    if (i < n &&
        (j == m || vi.vector_mul(vj) > eps))
      q = q + vi, i++;
    else
      q = q + vj, j++;
    result.push_back(q);
  }
  result.pop_back();
  return result;
}
```

Halfplanes Intersection $O(n \log n)$

```
template <class T> struct Q {
  T u = T(0);
  T v = T(1);
  // u / v
  // v > 0, gcd(|u|, |v|) = 1
  T gcd(T x, T y) {
    if (x < 0) x = -x;
    if (y < 0) y = -y;
    while (x) {
      // x, y -> y % x, x
      y %= x;
      swap(x, y);
    }
    return y;
  }
  Q() = default;
  Q(T uu, T vv = T(1)) {
    u = uu;
    v = vv;
    T g = gcd(uu, vv);
    u /= g;
    v /= g;
    if (v < 0) v = -v, u = -u;
  }
  Q operator+(const Q& o) const {
    return Q(u * o.v + o.u * v, v * o.v);
  }
  Q operator-(const Q& o) const {
    return Q(u * o.v - o.u * v, v * o.v);
  }
  Q operator*(const Q& o) const {
    return Q(u * o.u, v * o.v);
  }
  Q operator/(const Q& o) const {
    return Q(u * o.v, v * o.u);
  }
  bool operator==(const Q& o) const {
    return u * o.v == o.u * v;
  }
  bool operator<(const Q& o) const {
    return u * o.v < o.u * v;
  }
  bool operator>(const Q& o) const {
    return u * o.v > o.u * v;
  }
  bool operator<=(const Q& o) const {
    return u * o.v <= o.u * v;
  }
  bool operator>=(const Q& o) const {
    return u * o.v >= o.u * v;
  }
  ld to_ld() const { return ld(u) / ld(v); }
};
struct Line {
```

```
  ll a = 0;
  ll b = 0;
  ll c = 0;
  Line() = default;
  Line(ll aa, ll bb, ll cc) {
    a = aa;
    b = bb;
    c = cc;
    assert(a != 0 || b != 0);
  }
  template <class T>
  Q<T> vertical_line_to_x() const {
    return Q<T>(-c, a); // ax + c == 0, x = -c /
    ↪ a
  }
  bool parallel(const Line& o) const {
    return __int128(a) * o.b == __int128(b) *
    ↪ o.a;
  }
  template <class T>
  Q<T> get_x(const Line& o)
    const { // should not be parallel
    assert(!parallel(o));
    if (b == 0) return o.get_x<T>(*this);
    return Q<T>(
      T(o.b) * T(c) - T(o.c) * T(b),
      T(b) * T(o.a) -
        T(o.b) * T(a)); // (B2 * C1 - C2 * B1) /
                        // (B1 * A2 - B2 * A1)
  }
  ld get_y_by_x(ld x) const {
    return (-c - a * x) / b;
  }
  pair<ld, ld> intersect(const Line& o) const {
    ld x = get_x<__int128>(o).to_ld(), y;
    if (b)
      y = get_y_by_x(x);
    else
      y = o.get_y_by_x(x);
    return {x, y};
  }
  template <class T> Q<T> get_angle() const {
    return Q<T>(-a, b);
  }
  template <class T> Q<T> get_bias() const {
    return Q<T>(-c, b);
  }
  Line mirror_x() const { return {-a, b, c}; }
  Line mirror_y() const { return {a, -b, c}; }
};
struct Response {
  enum TYPE {
    INF,
    FINITE,
    EMPTY
  } type; // inf maybe in one or two directions
  vector<Line>
    lines; // lines in counter-clockwise order
};
vector<Line>
build_down_convex_hull(vector<Line> halfs) {
  sort(halfs.begin(), halfs.end(),
       [&](const Line& h1, const Line& h2) {
         __int128 hlp =
           __int128(-h1.a) * __int128(h2.b) -
           __int128(-h2.a) * __int128(h1.b);
         if (hlp == 0) {
           __int128 value =
             __int128(-h1.c) * __int128(h2.b) -
             __int128(-h2.c) * __int128(h1.b);
           if (h1.b < 0) value = -value;
           if (h2.b < 0) value = -value;
           return value < 0;
         }
         if (h1.b < 0) hlp = -hlp;
         if (h2.b < 0) hlp = -hlp;
         return hlp < 0;
       });
  vector<Line> st;
  for (Line L : halfs) {
    if ((int)st.size() >= 1 &&
        st.back().parallel(L))
      st.pop_back();
    while ((int)st.size() >= 2) {
      Line L1 = st[(int)st.size() - 2];
      Line L2 = st[(int)st.size() - 1];
      auto x1 = L1.get_x<__int128>(L2);
      auto x2 = L2.get_x<__int128>(L);
      if (x1 < x2) break;
      st.pop_back();
    }
    st.push_back(L);
  }
  return st;
}
template <class T>
void left_cut_hull(vector<Line>& hull, Q<T> LE) {
  int i = 0;
  while (i + 1 < (int)hull.size() &&
         hull[i].get_x<T>(hull[i + 1]) <= LE)
    i++;
  hull =
    vector<Line>(hull.begin() + i, hull.end());
}
vector<Line> concat_hulls(vector<Line> up,
                          vector<Line> down,
                          optional<Line> LE,
                          optional<Line> RI) {
  reverse(up.begin(), up.end());
  vector<Line> result;
  for (auto l : up)
    result.push_back(l);
  if (LE.has_value() &&
      (up.empty() || down.empty() ||
       up.back().get_angle<__int128>() <=
         down.front().get_angle<__int128>() ||
       up.back().get_x<__int128>(down.front()) <
         LE.value()
           .vertical_line_to_x<__int128>()))
    result.push_back(LE.value());
  for (auto l : down)
    result.push_back(l);
  if (RI.has_value() &&
      (up.empty() || down.empty() ||
       up.front().get_angle<__int128>() >=
```

```
        down.back().get_angle<__int128>() ||
      up.front().get_x<__int128>(down.back()) >
        RI.value()
          .vertical_line_to_x<__int128>()))
    result.push_back(RI.value());
  return result;
}
// ax + by + c >= 0, a^2 + b^2 > 0
// be careful with overfloating (|a,b,c| <= 10*9
// are ok, you can define __int128 ld) builds
// STRICTLY convex area (all unnecessary
// halfplanes will be ignored)
Response
halfplanes_intersection(vector<Line> halfs) {
  for (Line h : halfs)
    assert(h.a != 0 || h.b != 0);
  optional<Line> LE, RI;
  vector<Line> up, down;
  for (Line h : halfs) {
    if (h.b == 0) {  // vertical
      if (h.a > 0) { // to the right
        if (!LE.has_value() ||
            LE.value()
                .vertical_line_to_x<__int128>() <
              h.vertical_line_to_x<__int128>())
          LE = h;
      } else { // to the left
        if (!RI.has_value() ||
            RI.value()
                .vertical_line_to_x<__int128>() >
              h.vertical_line_to_x<__int128>())
          RI = h;
      }
    } else { // non-vertical
      if (h.b > 0)
        down.push_back(h);
      else
        up.push_back(h);
    }
  }
  if (LE.has_value() && RI.has_value() &&
      LE.value().vertical_line_to_x<__int128>() >

        ↪  RI.value().vertical_line_to_x<__int128>())
    return {Response::TYPE::EMPTY, {}};
  down = build_down_convex_hull(down);
  //    return {Response::TYPE::INF, {}};
  for (auto& l : up)
    l = l.mirror_y();
  up = build_down_convex_hull(up);
  for (auto& l : up)
    l = l.mirror_y();
  for (int phase = 0; phase < 2; phase++) {
    for (int iter = 0; iter < 2; iter++) {
      if (phase == 0) {
        if (LE.has_value()) {
          left_cut_hull<__int128>(
            down,
            LE.value()
              .vertical_line_to_x<__int128>());
          left_cut_hull<__int128>(
            up,
            LE.value()
              .vertical_line_to_x<__int128>());
        }
      } else {
        while (1) {
          int any = 0;
          if (!up.empty()) {
            int i = 0;
            while (
              i + 1 < (int)down.size() &&
              down[i + 1].get_angle<__int128>() <
                up[0].get_angle<__int128>() &&
              up[0].get_x<__int128>(down[i]) <=
                up[0].get_x<__int128>(
                  down[i + 1]))
              i++;
            any |= i > 0;
            down = vector<Line>(down.begin() + i,
                                down.end());
          }
          if (!down.empty()) {
            int i = 0;
            while (
              i + 1 < (int)up.size() &&
              up[i + 1].get_angle<__int128>() >
                down[0].get_angle<__int128>() &&
              down[0].get_x<__int128>(up[i]) <=
                down[0].get_x<__int128>(
                  up[i + 1]))
              i++;
            any |= i > 0;
            up = vector<Line>(up.begin() + i,
                              up.end());
          }
          if (!any) break;
        }
      }
      for (auto& l : up)
        l = l.mirror_x();
      for (auto& l : down)
        l = l.mirror_x();
      reverse(up.begin(), up.end());
      reverse(down.begin(), down.end());
      swap(LE, RI);
      if (LE.has_value())
        LE = LE.value().mirror_x();
      if (RI.has_value())
        RI = RI.value().mirror_x();
    }
  }
  vector<Line> result =
    concat_hulls(up, down, LE, RI);
  if (up.empty() || down.empty()) {
    return {Response::TYPE::INF, result};
  }
  if ((int)up.size() == 1 &&
      (int)down.size() == 1) {
    if (up[0].parallel(down[0])) {
      if (down[0].get_bias<__int128>() >
          up[0].get_bias<__int128>())
        return {Response::TYPE::EMPTY, {}};
      return {LE.has_value() && RI.has_value()
                ? Response::TYPE::FINITE
```

```
              : Response::TYPE::INF,
            result};
    } else {
      auto x0 = up[0].get_x<__int128>(down[0]);
      if (up[0].get_angle<__int128>() <
          down[0].get_angle<__int128>()) {
        if (LE.has_value() &&
            x0 <
              LE.value()
                .vertical_line_to_x<__int128>())
          return {Response::TYPE::EMPTY, {}};
      } else {
        if (RI.has_value() &&
            x0 >
              RI.value()
                .vertical_line_to_x<__int128>())
          return {Response::TYPE::EMPTY, {}};
      }
    }
  }
  bool is_empty = false;
  for (int iter = 0; iter < 2; iter++) {
    if ((int)down.size() >= 2 &&
        up[0].get_angle<__int128>() >
          down[0].get_angle<__int128>() &&
        up[0].get_x<__int128>(down[0]) >=
          down[0].get_x<__int128>(down[1]))
      is_empty = true;
    if ((int)down.size() >= 2 &&
        up[(int)up.size() - 1]
            .get_angle<__int128>() <
          down[(int)down.size() - 1]
            .get_angle<__int128>() &&
        up[(int)up.size() - 1].get_x<__int128>(
          down[(int)down.size() - 1]) <=
          down[(int)down.size() - 1]
            .get_x<__int128>(
              down[(int)down.size() - 2]))
      is_empty = true;
    for (auto& l : up)
      l = l.mirror_y();
    for (auto& l : down)
      l = l.mirror_y();
    swap(up, down);
  }
  if (is_empty)
    return {Response::TYPE::EMPTY, {}};
  auto type = Response::TYPE::FINITE;
  if (!LE.has_value() &&
      down.front().get_angle<__int128>() >=
        up.front().get_angle<__int128>())
    type = Response::TYPE::INF;
  if (!RI.has_value() &&
      down.back().get_angle<__int128>() <=
        up.back().get_angle<__int128>())
    type = Response::TYPE::INF;
  return {type, result};
}
ld halfplanes_intersection_area(
  Response response) {
  if (response.type == Response::TYPE::EMPTY)
    return 0;
  assert(response.type != Response::TYPE::INF);
  vector<pair<ld, ld>> p;
  auto lines = response.lines;
  int sz = (int)lines.size();
  ld area = 0;
  if (sz > 0) {
    for (int i = 0; i < sz; i++) {
      int j = (i + 1) % sz;
      p.push_back(lines[i].intersect(lines[j]));
    }
    for (int i = 0; i < sz; i++) {
      int j = (i + 1) % sz;
      auto [x1, y1] = p[i];
      auto [x2, y2] = p[j];
      area += x1 * y2 - x2 * y1;
    }
    area = max(area, ld(0));
  }
  return area / 2;
}
```

Fenwick Descent

```
struct Processor {
  int n = 0; // [0, n)
  vector<int> a;
  Processor() = default;
  Processor(int nn) {
    n = nn;
    a.assign(n, 0);
  }
  void increase(int i, int x) {
    for (int cur = i; cur < n; cur |= (cur + 1))
      a[cur] += x;
  }
  int descent(int lb) {
    int pos = 0;
    for (int pw = 1 << 19; pw > 0; pw >>= 1) {
      if (pos + pw <= n && a[pos + pw - 1] < lb)
      ↪ {
        lb -= a[pos + pw - 1];
        pos += pw;
      }
    }
    return pos;
  }
};
```

STL Tree

```
#include <ext/pb_ds/assoc_container.hpp>
#include <ext/pb_ds/tree_policy.hpp>
using namespace __gnu_pbds;
typedef tree<pair<int, int>, null_type,
↪ less<pair<int, int>>, rb_tree_tag,
             tree_order_statistics_node_update>
  stat_set;
```