

World Final Template

Mingyu Deng, Qipeng Kuang, Lihui Xie
Sun Yat-Sen University

1 Polynomials

```
1 namespace polynomial {
2
3 // dft.h
4 template <typename T>
5 class dft {
6 public:
7     static const bool use_fast_trans = true;
8     static void trans(std::vector<T>& p) {
9         assert(__builtin_popcount(p.size()) == 1);
10         if constexpr (use_fast_trans) {
11             dif(p);
12         } else {
13             bit_reverse(p);
14             dit(p);
15         }
16     }
17
18     static void inv_trans(std::vector<T>& p) {
19         assert(__builtin_popcount(p.size()) == 1);
20         if constexpr (use_fast_trans) {
21             dit(p);
22         } else {
23             trans(p);
24         }
25         reverse(p.begin() + 1, p.end());
26         T inv = T(p.size()).inv();
27         for (T& x : p) x *= inv;
28     }
29
30 // should call dit after dif
31 static void dit(std::vector<T>& p) {
32     for (int len = 1; len < p.size(); len <= 1) {
```

```
33         auto sub_w = get_subw(len * 2);
34         for (auto sub_p = p.begin(); sub_p != p.end(); sub_p += 2 * len)
35             for (int i = 0; i < len; ++i) {
36                 T u = sub_p[i], v = sub_p[i + len] * sub_w[i];
37                 sub_p[i] = u + v;
38                 sub_p[i + len] = u - v;
39             }
40     }
41 }
42
43 static void dif(std::vector<T>& p) {
44     for (int len = p.size() / 2; len >= 1; len >>= 1) {
45         auto sub_w = get_subw(len * 2);
46         for (auto sub_p = p.begin(); sub_p != p.end(); sub_p += 2 * len)
47             for (int i = 0; i < len; ++i) {
48                 T _sub_pi = sub_p[i];
49                 sub_p[i] += sub_p[i + len];
50                 sub_p[i + len] = (_sub_pi - sub_p[i + len]) * sub_w[i];
51             }
52     }
53 }
54
55 private:
56     typename std::vector<T>::iterator static get_subw(int len) {
57         static std::vector<T> w = {0, 1};
58         static const T primitive_root = T::primitive_root();
59         while (w.size() <= len) {
60             T e[] = {1, primitive_root.pow((T::modulus() - 1) / w.size())};
61             w.resize(w.size() * 2);
62             for (int i = w.size() / 2; i < w.size(); ++i) w[i] = w[i / 2] * e[i & 1];
63         }
64         return w.begin() + len;
65     }
66 };
67
68 // poly.h
69 template <typename T>
70 class poly : public std::vector<T> {
71 public:
72     using std::vector<T>::vector;
73
74     poly(std::string s) {
75         for (int i = 0; i < s.size(); i) {
76             auto scan_num = [&]() -> long long {
77                 int sgn = 1;
78                 if (s[i] == '-') sgn = -1, ++i;
79                 if (s[i] == '+') sgn = 1, ++i;
80                 if (i == s.size() || !std::isdigit(s[i])) return sgn;
81                 long long num = 0;
82                 while (i < s.size() && std::isdigit(s[i]))
83                     num = num * 10 + s[i++] - '0';
84                 return sgn * num;
85             };
```

```

85     };
86     auto add_item = [&](size_t exponent, T coeff) {
87         if (exponent >= this->size()) this->resize(exponent + 1);
88         this->at(exponent) = coeff;
89     };
90     T coeff = scan_num();
91     if (i == s.size() || s[i] != 'x')
92         add_item(0, coeff);
93     else {
94         size_t exponent = 1;
95         if (s[++i] == '^') {
96             ++i;
97             exponent = scan_num();
98         }
99         add_item(exponent, coeff);
100     }
101 }
102 }
103
104 int deg() const { return this->size() - 1; }
105 poly operator-() const {
106     poly ans = *this;
107     for (auto& x : ans) x = -x;
108     return ans;
109 }
110 T operator()(const T& x) const {
111     T ans = 0;
112     for (int i = this->size() - 1; i >= 0; --i) ans = ans * x + this->at(i);
113     return ans;
114 }
115 T operator[](int idx) const {
116     if (0 <= idx && idx < this->size()) return this->at(idx);
117     return 0;
118 }
119 T& operator[](int idx) {
120     if (idx >= this->size()) this->resize(idx + 1);
121     return this->at(idx);
122 }
123
124 poly rev() const {
125     poly res(*this);
126     std::reverse(res.begin(), res.end());
127     return res;
128 }
129
130 poly mulx(size_t k) const {
131     poly res = *this;
132     res.insert(res.begin(), k, 0);
133     return res;
134 }
135 poly divx(size_t k) const {
136     if (this->size() <= k) return {};

```

```

137     return poly(this->begin() + k, this->end());
138 }
139 poly modx(size_t k) const {
140     k = std::min(k, this->size());
141     return poly(this->begin(), this->begin() + k);
142 }
143
144 poly& operator*=(poly p) {
145     if (this->empty() || p.empty()) return *this = {};
146     constexpr int small_size = 128;
147     if (this->size() < small_size || p.size() < small_size) {
148         poly<T> t(this->size() + p.size() - 1);
149         for (int i = 0; i < this->size(); i++)
150             for (int j = 0; j < p.size(); j++) t[i + j] += this->at(i) * p[j];
151         return *this = t;
152     }
153     int len = 1 << (std::lg(this->deg() + p.deg()) + 1);
154     this->resize(len);
155     p.resize(len);
156     dft<T>::trans(*this);
157     dft<T>::trans(p);
158     for (int i = 0; i < len; ++i) this->at(i) *= p[i];
159     dft<T>::inv_trans(*this);
160     return this->normalize();
161 }
162
163 poly& operator+=(const poly& p) {
164     this->resize(std::max(this->size(), p.size()));
165     for (int i = 0; i < this->size(); ++i) this->at(i) += p[i];
166     return this->normalize();
167 }
168 poly& operator-=(const poly& p) {
169     this->resize(std::max(this->size(), p.size()));
170     for (int i = 0; i < this->size(); ++i) this->at(i) -= p[i];
171     return this->normalize();
172 }
173 poly& operator/=(const poly& p) {
174     if (this->size() < p.size()) return *this = {};
175     int len = this->size() - p.size() + 1;
176     return *this = (this->rev().modx(len) * p.rev().inv(len))
177         .modx(len)
178         .rev()
179         .normalize();
180 }
181 poly& operator%=(const poly& p) {
182     return *this = (*this - (*this / p) * p).normalize();
183 }
184 poly& operator+=(const T& x) {
185     for (int i = 0; i < this->size(); ++i) this->at(i) += x;
186     return *this;
187 }
188 poly& operator/=(const T& x) { return *this *= x.inv(); }

```

```

189 poly operator*(const poly& p) const { return poly(*this) * p; }
190 poly operator+(const poly& p) const { return poly(*this) + p; }
191 poly operator-(const poly& p) const { return poly(*this) - p; }
192 poly operator/(const poly& p) const { return poly(*this) / p; }
193 poly operator%(const poly& p) const { return poly(*this) % p; }
194 poly operator*(const T& x) const { return poly(*this) * x; }
195 poly operator/(const T& x) const { return poly(*this) / x; }
196
197 // (quotient, remainder)
198 std::pair<poly, poly> divmod(const poly& p) const {
199     poly d = *this / p;
200     return std::make_pair(d, (*this - d * p).normalize());
201 }
202
203 poly deriv() const {
204     if (this->empty()) return {};
205     poly res(this->size() - 1);
206     for (int i = 0; i < this->size() - 1; ++i) {
207         res[i] = this->at(i + 1) * (i + 1);
208     }
209     return res;
210 }
211
212 poly integr(T c = 0) const {
213     poly res(this->size() + 1);
214     for (int i = 0; i < this->size(); ++i) {
215         res[i + 1] = this->at(i) / (i + 1);
216     }
217     res[0] = c;
218     return res;
219 }
220
221 // mod x^k
222 poly inv(int k = -1) const {
223     if (!k) k = this->size();
224     poly res = {this->front().inv()};
225     for (int len = 2; len < k * 2; len <= 1) {
226         res = (res * (poly{2} - this->modxx(len) * res)).modxx(len);
227     }
228     return res.modxx(k);
229 }
230
231 // mod x^k
232 poly sqrt(int k = -1) const {
233     if (!k) k = this->size();
234     poly res = {this->at(0).sqrt()};
235     for (int len = 2; len < k * 2; len <= 1) {
236         res = (res + (this->modxx(len) * res.inv(len)).modxx(len)) / 2;
237     }
238     return res.modxx(k);
239 }
240
241 // mod x^k, a0=1 should hold

```

```

241 poly log(int k = -1) const {
242     assert(this->at(0) == 1);
243     if (!k) k = this->size();
244     return (this->deriv() * this->inv(k)).integr().modxx(k);
245 }
246
247 // mod x^k, a0=0 should hold
248 poly exp(int k = -1) const {
249     assert(this->at(0) == 0);
250     if (!k) k = this->size();
251     poly res = {1};
252     for (int len = 2; len < k * 2; len <= 1) {
253         res = (res * (poly{1} - res.log(len) + this->modxx(len))).modxx(len);
254     }
255     return res.modxx(k);
256 }
257
258 // p^c mod x^k
259 poly pow(int c, int k = -1) const {
260     if (!k) k = this->size();
261     int i = 0;
262     while (i < this->size() && !this->at(i)) ++i;
263     if (i == this->size() || 1LL * i * c >= k) return {};
264     T ai = this->at(i);
265     poly f = this->divxx(i) * ai.inv();
266     return (f.log(k - i * c) * c).exp(k - i * c).mulxx(i * c) * ai.pow(c);
267 }
268
269 // evaluate and interpolate
270 struct product_tree {
271     int l, r;
272     std::unique_ptr<product_tree> lson = nullptr, rson = nullptr;
273     poly product;
274     product_tree(int l, int r) : l(l), r(r) {}
275
276     static std::unique_ptr<product_tree> build(
277         const std::vector<T>& xs, std::function<poly(T)> get_poly) {
278         std::function<std::unique_ptr<product_tree>(int, int)> build =
279             [&](int l, int r) {
280                 auto rt = std::make_unique<product_tree>(l, r);
281                 if (l == r) {
282                     rt->product = get_poly(xs[l]);
283                 } else {
284                     int mid = (l + r) >> 1;
285                     rt->lson = build(l, mid);
286                     rt->rson = build(mid + 1, r);
287                     rt->product = rt->lson->product * rt->rson->product;
288                 }
289                 return rt;
290             };
291         return build(0, xs.size() - 1);
292 }

```

```

293 poly mulT(poly p) const {
294     if (p.empty()) return {};
295     return ((*this) * p.rev()).divxk(p.size() - 1);
296 }
297
298 std::vector<T> evaluate(std::vector<T> xs) const {
299     if (this->empty()) return std::vector<T>(xs.size());
300     std::unique_ptr<product_tree> rt = product_tree::build(xs, [&](T x) {
301         return poly{1, -x};
302     });
303     return evaluate_internal(xs, rt);
304 }
305
306 static poly interpolate(std::vector<T> xs, std::vector<T> ys) {
307     assert(xs.size() == ys.size());
308     if (xs.empty()) return {};
309     std::unique_ptr<product_tree> rt = product_tree::build(xs, [&](T x) {
310         return poly{1, -x};
311     });
312     std::vector<T> coef = rt->product.rev().deriv().evaluate_internal(xs, rt);
313     for (int i = 0; i < ys.size(); ++i) coef[i] = ys[i] * coef[i].inv();
314     std::function<poly(product_tree*)> solve = [&](product_tree* rt) {
315         if (rt->l == rt->r) {
316             return poly{coef[rt->l]};
317         } else {
318             return solve(rt->lson.get()) * rt->rson->product.rev() +
319                 solve(rt->rson.get()) * rt->lson->product.rev();
320         }
321     };
322     return solve(rt.get());
323 }
324
325 // a0=0 must hold
326 poly cos(int k = -1) const {
327     assert(this->at(0) == 0);
328     if (!k) k = this->size();
329     T i = T::root().pow((T::modulus() - 1) / 4);
330     poly x = *this * i;
331     return (x.exp(k) + (-x).exp(k)) / 2;
332 }
333
334 // a0=0 must hold
335 poly sin(int k = -1) const {
336     assert(this->at(0) == 0);
337     if (!k) k = this->size();
338     T i = T::root().pow((T::modulus() - 1) / 4);
339     poly x = *this * i;
340     return (x.exp(k) - (-x).exp(k)) / (i * 2);
341 }
342
343 // a0=0 must hold
344 poly tan(int k = -1) const { return this->sin(k) / this->cos(k); }

```

```

345
346 poly acos(int k = -1) const {
347     const poly& x = *this;
348     return (-x.deriv() * (poly{1} - x * x).sqrt().inv()).integr();
349 };
350
351 poly asin(int k = -1) const {
352     const poly& x = *this;
353     return (x.deriv() * (poly{1} - x * x).sqrt().inv()).integr();
354 };
355
356 poly atan(int k = -1) const {
357     const poly& x = *this;
358     return (x.deriv() * (poly{1} + x * x).inv()).integr();
359 };
360
361 friend std::ostream& operator<<(std::ostream& os, poly p) {
362     os << "{";
363     for (auto x : p) os << x() << " ";
364     os << "}";
365     return os;
366 }
367
368 private:
369 poly& normalize() {
370     while (this->size() && !this->back() this->pop_back());
371     return *this;
372 }
373
374 std::vector<T> evaluate_internal(std::vector<T>& xs,
375     std::unique_ptr<product_tree>& rt) const {
376     std::vector<T> res(xs.size());
377     xs.resize(std::max(xs.size(), this->size()));
378     std::function<void(product_tree*, poly)> solve = [&](product_tree* rt,
379         poly p) {
380         p = p.modxk(rt->r - rt->l + 1);
381         if (rt->l == rt->r) {
382             if (rt->l < res.size()) res[rt->l] = p.front();
383         } else {
384             solve(rt->lson.get(), p.mulT(rt->rson->product));
385             solve(rt->rson.get(), p.mulT(rt->lson->product));
386         }
387     };
388     solve(rt.get(), this->mulT(rt->product.inv(xs.size())));
389     return res;
390 }
391
392 // namespace polynomial
393 using poly = polynomial::poly<zint>;

```

2 Strings

```

1  //***** Lyndon *****
2
3  namespace lyndon {
4
5  std::vector<int> getfactorization(const std::string &s) {
6      std::vector<int> right_ends;
7      for (int i = 0; i < s.length(); ) {
8          int j = i, k = i + 1;
9          for (; k < s.length() && s[j] <= s[k]; j++, k++)
10             if (s[j] < s[k]) j = i - 1;
11             while (i <= j) i += k - j, right_ends.push_back(i);
12         }
13         return right_ends;
14     }
15 }
16 // namespace lyndon
17
18 //***** Pam *****
19
20 template <size_t alphabet_size = 26>
21 class palindrome_automaton {
22 public:
23     struct node {
24         std::array<int, alphabet_size> to;
25         int link, len, count;
26
27         explicit node(int len = 0) : len(len), link(-1), count(0) { to.fill(0); }
28         explicit node(int len, int link) : len(len), link(link), count(0) {
29             to.fill(0);
30         }
31     };
32
33     palindrome_automaton() {
34         int even_rt = newnode(0);
35         int odd_rt = newnode(-1);
36         nodes[even_rt].link = odd_rt;
37         nodes[odd_rt].link = even_rt;
38         last = even_rt;
39     }
40
41     void extend(char c) {
42         text.push_back(c);
43         int i = text.size() - 1;
44         auto getlink = [&](int u) {
45             while (i - nodes[u].len - 1 < 0 || text[i - nodes[u].len - 1] != c) {
46                 u = nodes[u].link;
47             }
48             return u;
49         };

```

```

50         int w = c - 'a';
51         int u = getlink(last);
52         if (!nodes[u].to[w]) {
53             int v = newnode(nodes[u].len + 2, nodes[getlink(nodes[u].link)].to[w]);
54             nodes[u].to[w] = v;
55         }
56         last = nodes[u].to[w];
57         ++nodes[last].count; // should be accumulated later from fail link tree
58     }
59
60     std::string to_string() const {
61         std::ostringstream os;
62         std::function<void(int, std::string)> travel = [&](int k, std::string s) {
63             if (!k) return;
64             os << k << " "; s << s << " - " << nodes[k].count << "\n";
65             for (int c = 0; c < alphabet_size; ++c)
66                 travel(nodes[k].to[c], std::string(1, c + 'a') + s + (char)(c + 'a'));
67         };
68         for (int c = 0; c < alphabet_size; ++c)
69             travel(nodes[0].to[c], std::string(2, c + 'a'));
70         for (int c = 0; c < alphabet_size; ++c)
71             travel(nodes[1].to[c], std::string(1, c + 'a'));
72         return os.str();
73     }
74
75 private:
76     int last;
77     std::string text;
78     std::vector<node> nodes;
79
80     template <typename... Args>
81     int newnode(Args... args) {
82         int res = nodes.size();
83         nodes.push_back(node{args...});
84         return res;
85     }
86 };
87
88 //***** Suffix Array *****
89
90 template <typename Container = std::vector<int>>
91 struct SuffixArray {
92     int n;
93     Container s;
94     // lc[0]=0 is meaningless
95     std::vector<int> sa, rk, lc;
96     SuffixArray(const Container& s)
97         : s(s), n(s.size()), sa(s.size()), rk(s.size()), lc(s.size()) {
98         std::iota(sa.begin(), sa.end(), 0);
99         std::sort(sa.begin(), sa.end(), [&](int a, int b) { return s[a] < s[b]; });
100         rk[sa[0]] = 0;
101         for (int i = 1; i < n; ++i)

```

```

102     rk[sa[i]] = rk[sa[i - 1]] + (s[sa[i]] != s[sa[i - 1]]);
103     std::vector<int> tmp, cnt(n);
104     tmp.reserve(n);
105     for (int k = 1; rk[sa[n - 1]] < n - 1; k *= 2) {
106         tmp.clear();
107         for (int i = 0; i < k; ++i) tmp.push_back(n - k + i);
108         for (auto i : sa)
109             if (i >= k) tmp.push_back(i - k);
110         cnt.assign(n, 0);
111         for (int i = 0; i < n; ++i) ++cnt[rk[i]];
112         for (int i = 1; i < n; ++i) cnt[i] += cnt[i - 1];
113         for (int i = n - 1; i >= 0; --i) sa[--cnt[rk[tmp[i]]]] = tmp[i];
114         std::swap(rk, tmp);
115         rk[sa[0]] = 0;
116         for (int i = 1; i < n; ++i) {
117             rk[sa[i]] = rk[sa[i - 1]];
118             if (tmp[sa[i - 1]] < tmp[sa[i]] || sa[i - 1] + k == n ||
119                 tmp[sa[i - 1] + k] < tmp[sa[i] + k])
120                 ++rk[sa[i]];
121         }
122     }
123     for (int i = 0, j = 0; i < n; ++i) {
124         if (!rk[i]) {
125             j = 0;
126         } else {
127             if (j) --j;
128             int k = sa[rk[i] - 1];
129             while (i + j < n && k + j < n && s[i + j] == s[k + j]) ++j;
130             lc[rk[i]] = j;
131         }
132     }
133 }
134 };
135
136 template <typename Container = std::vector<int>>
137 class LongestCommonPrefix {
138 public:
139     LongestCommonPrefix(SuffixArray<Container>* sa) : sa(sa), st(sa->lc) {}
140
141     int lcp(int i, int j) {
142         assert(0 <= i && i <= sa->n);
143         assert(0 <= j && j <= sa->n);
144         if (i == sa->n || j == sa->n) return 0;
145         if (i == j) return sa->n - i;
146         int l = sa->rk[i], r = sa->rk[j];
147         if (l > r) std::swap(l, r);
148         return st.queryMin(l + 1, r);
149     }
150
151 private:
152     SuffixArray<Container>* sa;
153     SparseTable<int> st;

```

```

154 };
155
156 //***** SAM *****
157
158 template <size_t alphabet_size = 26>
159 class suffix_automaton {
160 public:
161     struct node {
162         std::array<int, alphabet_size> to;
163         int link, len, count;
164
165         explicit node(int len = 0) : len(len), link(-1), count(0) { to.fill(-1); }
166         explicit node(int len, int link, const std::array<int, alphabet_size>& to)
167             : len(len), link(link), to(to), count(0) {}
168     };
169
170     suffix_automaton() : nodes() { newnode(); }
171
172     explicit suffix_automaton(const std::string& s) : suffix_automaton() {
173         insert(s);
174     }
175
176     void insert(const std::string& s) {
177         nodes.reserve(size() + s.size() * 2);
178         int last = 0;
179         for (int i = 0; i < s.size(); ++i) {
180             last = extend(last, s[i] - 'a');
181         }
182     }
183
184     int extend(int k, int c) {
185         if (~nodes[k].to[c] && nodes[nodes[k].to[c]].len == nodes[k].len + 1) {
186             return nodes[k].to[c];
187         }
188         int leaf = newnode(nodes[k].len + 1);
189         for (; ~k && !nodes[k].to[c]; k = nodes[k].link) nodes[k].to[c] = leaf;
190         if (!k) {
191             nodes[leaf].link = 0;
192         } else {
193             int p = nodes[k].to[c];
194             if (nodes[k].len + 1 == nodes[p].len) {
195                 nodes[leaf].link = p;
196             } else {
197                 int np = newnode(nodes[k].len + 1, nodes[p].link, nodes[p].to);
198                 nodes[p].link = nodes[leaf].link = np;
199                 for (; ~k && nodes[k].to[c] == p; k = nodes[k].link)
200                     nodes[k].to[c] = np;
201             }
202         }
203         return leaf;
204     }
205

```

```

206 void build_ancestors() {
207     for (int i = 1; i < size(); ++i) ancestors[i] = {nodes[i].link};
208     for (int j = 1; (1 << j) < size(); ++j) {
209         for (int i = 0; i < size(); ++i)
210             if (~ancestors[i][j - 1]) {
211                 ancestors[i][j] = ancestors[ancestors[i][j - 1]][j - 1];
212             } else {
213                 ancestors[i][j] = -1;
214             }
215     }
216 }
217
218 std::vector<int> mark_count(const std::string& s) {
219     std::vector<int> ends;
220     int k = 0;
221     for (char c : s) {
222         k = nodes[k].to[c - 'a'];
223         assert(~k);
224         ends.push_back(k);
225         ++nodes[k].count;
226     }
227     return ends;
228 }
229
230 void addup_count() {
231     std::vector<int> ids(size()), bucket(size());
232     for (int i = 0; i < size(); ++i) ++bucket[nodes[i].len];
233     for (int i = 1; i < bucket.size(); ++i) bucket[i] += bucket[i - 1];
234     for (int i = 0; i < size(); ++i) ids[--bucket[nodes[i].len]] = i;
235     for (int i = size() - 1; i; --i)
236         nodes[nodes[ids[i]].link].count += nodes[ids[i]].count;
237 }
238
239 const node& operator[](int v) const { return nodes[v]; }
240
241 int maxlen(int v) const { return nodes[v].len; }
242
243 int minlen(int v) const { return v ? nodes[nodes[v].link].len + 1 : 0; }
244
245 int size() const { return nodes.size(); }
246
247 std::string to_string() const {
248     std::ostringstream os;
249     std::function<void(int, std::string)> travel = [&](int k, std::string s) {
250         if (!k) return;
251         os << k << ": " << s << " - " << nodes[k].count << "\n";
252         for (int c = 0; c < alphabet_size; ++c)
253             travel(nodes[k].to[c], s + (char)(c + 'a'));
254     };
255     travel(0, "");
256     return os.str();
257 }

```

```

258
259 private:
260     std::vector<node> nodes;
261     std::vector<std::vector<int>> ancestors;
262
263     template <typename... Args>
264     int newnode(Args... args) {
265         int res = nodes.size();
266         nodes.push_back(node{args...});
267         return res;
268     }
269 };
270
271 //***** ACauto *****
272
273 int go[maxtri][26], sum, count[maxtri], d[maxtri], fail[maxtri];
274 void make_tri(int k) {
275     fo(i, 0, nb-1) {
276         int index=sb[i]- 'a';
277         if (!go[k][index]) go[k][index]=++sum;
278         k=go[k][index];
279     }
280     count[k]++;
281 }
282 void make_fail() {
283     int i=0, j=0;
284     fo(p, 0, 25) if (go[0][p]) d[++j]=go[0][p];
285     while (i++<j) {
286         int now=d[i];
287         fo(p, 0, 25) if (go[now][p]) {
288             int son=go[now][p];
289             fail[son]=go[fail[now]][p];
290             count[son]+=count[fail[son]];
291             d[++j]=son;
292         } else go[now][p]=go[fail[now]][p];
293     }
294 }
295 void find(int k) { //sa中出现了多少次sb
296     fo(i, 1, n) ans+=count[k=go[k][sa[i]- 'a']];
297 }
298
299 //***** manacher *****
300
301 int f[2*maxn];
302 void manacher()
303 {
304     int lim=0, mid=0;
305     fo(i, 1, m) // m=2*n+1
306     {
307         f[i]= (i<=lim) ? min(f[mid*2-i], lim-i+1) : 1 ;
308         while (i-f[i]>0 && i+f[i]<=m && s[i-f[i]]==s[i+f[i]]) f[i]++;
309         if (i+f[i]-1>lim) lim=i+f[i]-1, mid=i;

```

```

310     }
311 }
312
313 //***** exkmp *****
314
315 int next[maxn], ex[maxn];
316 void exkmp() {
317     next[1]=nb;
318     int k=0;
319     fo(i,2,nb) {
320         int lim=k+next[k]-1, L=next[i-k+1];
321         if (i+L<=lim) next[i]=L; else {
322             next[i]=max(lim-i+1,0);
323             while (i+next[i]<=nb && sb[i+next[i]]==sb[1+next[i]]) next[i]++;
324             k=i;
325         }
326     }
327
328     k=1;
329     fo(i,1,na) {
330         int lim=k+ex[k]-1, L=next[i-k+1];
331         if (i+L<=lim) ex[i]=L; else {
332             ex[i]=max(lim-i+1,0);
333             while (i+ex[i]<=na && 1+ex[i]<=nb && sa[i+ex[i]]==sb[1+ex[i]]) ex[i]++;
334             k=i;
335         }
336     }
337 }
338
339 //***** min_representation *****
340
341 int s[maxn];
342 int min_representation(int *s,int len) { // index from 0
343     int i=0, j=1;
344     while (i<len && j<len) {
345         int k=0;
346         for(; k<len && s[(i+k)%len]==s[(j+k)%len]; k++);
347         if (k==len) break;
348         (s[(i+k)%len]<s[(j+k)%len]) ? j+=k+1 : i+=k+1;
349         i+=(i==j);
350     }
351     return min(i,j);
352 }

```

3 Geometry

```

1 namespace geometry2d {
2

```

```

3 constexpr long double eps = 1e-7;
4 constexpr long double pi = std::acos(-1);
5
6 int fsign(long double x) { return (x > eps) - (x < -eps); }
7 long double sqr(long double x) { return x * x; }
8
9 // Ax^2+Bx+c=0
10 std::vector<long double> solveEquationP2(long double A, long double B,
11                                         long double C) {
12     long double delta = B * B - 4 * A * C;
13     if (fsign(delta) < 0) return {};
14     if (fsign(delta) == 0) return {-0.5 * B / A};
15     long double sqrt_delta = std::sqrt(delta);
16     long double x1 = -0.5 * (B - sqrt_delta) / A,
17                 x2 = -0.5 * (B + sqrt_delta) / A;
18     if (fsign(x1 - x2) > 0) std::swap(x1, x2);
19     return {x1, x2};
20 }
21
22 struct Point;
23 struct Point3D {
24     long double x, y, z;
25
26     explicit Point3D(long double x = 0, long double y = 0, long double z = 0)
27         : x(x), y(y), z(z) {}
28
29     explicit Point3D(const Point& p);
30
31     Point3D operator-(const Point3D& p) const {
32         return Point3D(x - p.x, y - p.y, z - p.z);
33     }
34     long double innerProd(const Point3D& p) const {
35         return x * p.x + y * p.y + z * p.z;
36     }
37     Point3D crossProd(const Point3D& p) const {
38         return Point3D(y * p.z - z * p.y, -x * p.z + z * p.x, x * p.y - y * p.x);
39     }
40 };
41
42 struct Point {
43     long double x, y;
44     explicit Point(long double x = 0, long double y = 0) : x(x), y(y) {}
45     Point operator+(const Point& p) const { return Point(x + p.x, y + p.y); }
46     Point operator-(const Point& p) const { return Point(x - p.x, y - p.y); }
47     Point operator/(const long double& l) const { return Point(x / l, y / l); }
48     Point operator*(const long double& l) const { return Point(x * l, y * l); }
49     Point unit() const {
50         long double l = len();
51         assert(fsign(l) > 0);
52         return *this / l;
53     }
54     long double crossProd(const Point& p) const { return x * p.y - y * p.x; }

```



```

55 long double innerProd(const Point& p) const { return x * p.x + y * p.y; }
56 long double lenSqr() const { return sqr(x) + sqr(y); }
57 long double len() const { return std::sqrt(lenSqr()); }
58 long double distanceSqr(const Point& p) const {
59     return sqr(x - p.x) + sqr(y - p.y);
60 }
61 long double distance(const Point& p) const {
62     return std::sqrt(distanceSqr(p));
63 }
64 long double angleWith(const Point& p) const {
65     return std::atan2(crossProd(p), innerProd(p));
66 }
67 // return  $(-\pi, \pi]$ 
68 long double angle() const { return std::atan2(y, x); }
69 int angleSign() const {
70     if (!y) return fsign(x) < 0;
71     return fsign(y);
72 }
73 Point rotate90() const { return Point(-y, x); }
74
75 static long double crossProd(const Point& o, const Point& a, const Point& b) {
76     return (a - o).crossProd(b - o);
77 }
78 static long double innerProd(const Point& o, const Point& a, const Point& b) {
79     return (a - o).innerProd(b - o);
80 }
81
82 friend std::ostream& operator<<(std::ostream& os, Point p) {
83     return os << "(" << p.x << ", " << p.y << ")";
84 }
85
86 friend bool operator==(const Point& lhs, const Point& rhs) {
87     return !fsign(lhs.x - rhs.x) && !fsign(lhs.y - rhs.y);
88 }
89
90 struct HorizontalComparer {
91     bool operator()(const Point& lhs, const Point& rhs) const {
92         return lhs.x < rhs.x || (lhs.x == rhs.x && lhs.y < rhs.y);
93     }
94 };
95
96 struct PolarComparer {
97     bool operator()(const Point& lhs, const Point& rhs) const {
98         int angleSign_diff = lhs.angleSign() - rhs.angleSign();
99         if (angleSign_diff) return angleSign_diff < 0;
100         int crossProd_sign = fsign(lhs.crossProd(rhs));
101         if (crossProd_sign) return crossProd_sign > 0;
102         return lhs.lenSqr() < rhs.lenSqr();
103     }
104 };
105 };
106

```

```

107 Point3D::Point3D(const Point& p) : Point3D(p.x, p.y, p.x * p.x + p.y * p.y) {}
108
109 struct Line {
110     Point pivot, unit_direction;
111     Line(Point pivot, Point direction)
112         : pivot(pivot), unit_direction(direction.unit()) {}
113     long double angle() const { return unit_direction.angle(); }
114     // 1: on the left
115     // 0: in the line
116     // -1: on the right
117     int side(const Point& p) {
118         return fsign(unit_direction.crossProd(p - pivot));
119     }
120     bool isParallel(const Line& l) {
121         return fsign(unit_direction.crossProd(l.unit_direction)) == 0;
122     }
123     bool isSame(const Line& l) {
124         return isParallel(l) &&
125             fsign(unit_direction.crossProd(l.pivot - pivot)) == 0;
126     }
127     bool isSameDirection(const Line& l) {
128         return isParallel(l) &&
129             fsign(unit_direction.innerProd(l.unit_direction)) > 0;
130     }
131     Point intersection(const Line& l) {
132         assert(!isParallel(l));
133         return pivot + unit_direction *
134             (l.unit_direction.crossProd(l.pivot - pivot)) /
135             l.unit_direction.crossProd(unit_direction);
136     }
137
138     friend std::ostream& operator<<(std::ostream& os, Line l) {
139         return os << "(" << l.pivot << ", " << l.unit_direction << ")";
140     }
141 };
142
143 struct Segment {
144     Point a, b;
145     Segment(Point a, Point b) : a(a), b(b) {}
146     bool hasIntersection(const Segment& s) const {
147         return fsign(Point::crossProd(a, b, s.a)) *
148             fsign(Point::crossProd(a, b, s.b)) <
149             0 &&
150             fsign(Point::crossProd(s.a, s.b, a)) *
151             fsign(Point::crossProd(s.a, s.b, b)) <
152             0;
153     }
154     bool isPointIn(const Point& p) const {
155         return fsign(Point::innerProd(p, a, b)) <= 0 &&
156             fsign(Point::crossProd(p, a, b)) == 0;
157     }
158

```

```

159 Point lerp(const long double& ratio) const { return a + (b - a) * ratio; }
160
161 friend std::ostream& operator<<(std::ostream& os, Segment s) {
162     return os << "(" << s.a << ", " << s.b << ")";
163 }
164 };
165
166 struct Polygon {
167     static bool isConvexInCCW(const std::vector<Point>& p) {
168         for (int i = 0; i < p.size(); ++i) {
169             int l = (i + p.size() - 1) % p.size();
170             int r = (i + 1) % p.size();
171             if (Point::crossProd(p[i], p[l], p[r]) > 0) return false;
172         }
173         return true;
174     }
175     static std::vector<int> convexHullId(const std::vector<Point>& p) {
176         if (p.size() == 0) return {};
177         if (p.size() == 1) return {0};
178         std::vector<int> ids(p.size());
179         std::iota(ids.begin(), ids.end(), 0);
180         sort(ids.begin(), ids.end(),
181             [& comp = Point::HorizontalComparer()](int i, int j) {
182                 return comp(p[i], p[j]);
183             });
184         std::vector<int> res;
185         for (int i : ids) {
186             while (res.size() > 1 &&
187                 fsign(Point::crossProd(p[res.end()[-2]], p[res.end()[-1]],
188                     p[i])) <= 0)
189                 res.pop_back();
190             res.push_back(i);
191         }
192         ids.pop_back();
193         std::reverse(ids.begin(), ids.end());
194         int lower_size = res.size();
195         for (int i : ids) {
196             while (res.size() > lower_size &&
197                 fsign(Point::crossProd(p[res.end()[-2]], p[res.end()[-1]],
198                     p[i])) <= 0)
199                 res.pop_back();
200             res.push_back(i);
201         }
202         res.pop_back();
203         return res;
204     }
205     static std::vector<Point> convexHullPoint(const std::vector<Point>& p) {
206         std::vector<int> ids = convexHullId(p);
207         std::vector<Point> res;
208         for (int i : ids) res.push_back(p[i]);
209         return res;
210     }

```

```

211
212 // should be guaranteed that convex is a convex hull in ccw
213 static bool isPointInConvexCCW(const Point& p,
214     const std::vector<Point>& convex) {
215     assert(Polygon::isConvexInCCW(convex));
216     for (int i = 0; i < convex.size(); ++i) {
217         Point a = convex[i];
218         Point b = convex[(i + 1) % convex.size()];
219         if (Point::crossProd(a, b, p) < 0) return false;
220     }
221     return true;
222 }
223 };
224
225 struct Circle {
226     Point o;
227     long double r;
228     Circle() : Circle(Point(0, 0), 0) {}
229     Circle(Point o, long double r) : o(o), r(r) {}
230     Point pointInDirection(long double angle) {
231         return Point(o.x + r * std::cos(angle), o.y + r * std::sin(angle));
232     }
233
234     std::vector<Point> intersection(const Line& l) const {
235         long double A = 1;
236         long double B = l.unit_direction.innerProd(l.pivot - o) * 2;
237         long double C = o.distanceSqr(l.pivot) - sqr(r);
238         std::vector<long double> roots = solveEquationP2(A, B, C);
239         std::vector<Point> intersects;
240         for (long double x : roots)
241             intersects.push_back(l.pivot + l.unit_direction * x);
242         return intersects;
243     }
244
245     std::vector<Point> intersection(const Segment& s) const {
246         std::vector<Point> line_intersects = intersection(Line(s.a, s.b - s.a));
247         std::vector<Point> intersects;
248         for (const Point& p : line_intersects)
249             if (s.isPointIn(p)) {
250                 intersects.push_back(p);
251             }
252         return intersects;
253     }
254
255 // triangle oab
256 long double overlapAreaWithTriangle(const Point& a, const Point& b) const {
257     if (!fsign(Point::crossProd(o, a, b))) return 0;
258     std::vector<Point> key_points;
259     key_points.push_back(a);
260     for (const Point& p : intersection(Segment(a, b))) key_points.push_back(p);
261     key_points.push_back(b);
262     long double res = 0;
263     for (int i = 1; i < key_points.size(); ++i) {

```

```

263     Point mid_point = (key_points[i - 1] + key_points[i]) / 2;
264     Point ray1 = key_points[i - 1] - o;
265     Point ray2 = key_points[i] - o;
266     if (o.distanceSqr(mid_point) <= sqr(r)) {
267         res += std::abs(ray1.crossProd(ray2));
268     } else {
269         res += sqr(r) * std::abs(ray1.angleWith(ray2));
270     }
271 }
272 dbg(a, b, key_points, res);
273 return 0.5 * res;
274 }
275
276 long double overlapAreaWithPolygon(const std::vector<Point>& p) const {
277     long double res = 0;
278     for (int i = 0; i < p.size(); ++i) {
279         int j = (i + 1) % p.size();
280         res += overlapAreaWithTriangle(p[i], p[j]) *
281             fsign(Point::crossProd(o, p[i], p[j]));
282     }
283     dbg(res);
284     return res;
285 }
286
287 // 1: outside
288 // 0: edge
289 // -1: inside
290 // equal to solving following determinant (a,b,c is counter-clockwise)
291 // | ax, ay, ax^2+ay^2, 1 |
292 // | bx, by, bx^2+by^2, 1 |
293 // | cx, cy, cx^2+cy^2, 1 |
294 // | px, py, px^2+py^2, 1 |
295 static int side(const Point& a, Point b, Point c, const Point& p) {
296     if (fsign(Point::crossProd(a, b, c)) < 0) std::swap(b, c);
297     Point3D a3(a), b3(b), c3(c), p3(p);
298     b3 = b3 - a3;
299     c3 = c3 - a3;
300     p3 = p3 - a3;
301     Point3D f = b3.crossProd(c3);
302     return fsign(p3.innerProd(f));
303 }
304
305 friend std::ostream& operator<<(std::ostream& os, const Circle& c) {
306     return os << "(" << c.o << ", " << c.r << ")";
307 }
308 };
309
310 // return the intersection convex in ccw, should be guaranteed that the
311 // intersection is finite.
312 struct HalfPlaneIntersection {
313     static std::vector<Point> solve(std::vector<Line> lines) {
314         sort(lines.begin(), lines.end(),
315
316     [comp = Point::PolarComparer()](auto l1, auto l2) {
317         if (l1.isSameDirection(l2)) {
318             return l1.side(l2.pivot) < 0;
319         } else {
320             return comp(l1.unit_direction, l2.unit_direction);
321         }
322     });
323     std::deque<Line> key_lines;
324     std::deque<Point> key_points;
325     for (int i = 0; i < lines.size(); ++i) {
326         if (i > 0 && lines[i - 1].isSameDirection(lines[i])) continue;
327         while (key_points.size() && lines[i].side(key_points.back()) <= 0) {
328             key_lines.pop_back();
329             key_points.pop_back();
330         }
331         while (key_points.size() && lines[i].side(key_points.front()) <= 0) {
332             key_lines.pop_front();
333             key_points.pop_front();
334         }
335         if (key_lines.size() {
336             // since it's guaranteed that the intersection is finite, therefore must
337             // be empty.
338             if (lines[i].isParallel(key_lines.back())) return {};
339             key_points.push_back(lines[i].intersection(key_lines.back()));
340         }
341         key_lines.push_back(lines[i]);
342     }
343
344     while (key_points.size() &&
345            key_lines.front().side(key_points.back()) <= 0) {
346         key_lines.pop_back();
347         key_points.pop_back();
348     }
349
350     if (key_lines.size() <= 2) return {};
351
352     std::vector<Point> convex;
353     for (int i = 0; i < key_lines.size(); ++i)
354         convex.emplace_back(
355             key_lines[i].intersection(key_lines[(i + 1) % key_lines.size()]));
356     return convex;
357 }
358 };
359
360 struct Triangulation {
361     struct Edge {
362         int v;
363         std::list<Edge>::iterator rev;
364         Edge(int v = 0) : v(v) {}
365     };
366     // delaunay triangulation

```

```

367 // should be guaranteed that all points are pairwise distinct
368 static std::vector<std::pair<int, int>> nearest(const std::vector<Point>& p) {
369     std::vector<std::list<Edge>> neighbor(p.size());
370     std::vector<int> id(p.size());
371     std::iota(id.begin(), id.end(), 0);
372     std::sort(id.begin(), id.end(),
373         [&, comp = Point::HorizontalComparer()](int i, int j) {
374             return comp(p[i], p[j]);
375         });
376
377     auto addedge = [&](int u, int v) {
378         neighbor[u].push_front(v);
379         neighbor[v].push_front(u);
380         neighbor[u].front().rev = neighbor[v].begin();
381         neighbor[v].front().rev = neighbor[u].begin();
382     };
383     std::function<void(int, int)> divide = [&](int l, int r) {
384         if (r - l + 1 <= 3) {
385             for (int i = l; i <= r; ++i)
386                 for (int j = l; j < i; ++j) addedge(id[i], id[j]);
387             return;
388         }
389
390         int mid = (l + r) >> 1;
391         divide(l, mid);
392         divide(mid + 1, r);
393
394         auto get_base_LR_edge = [&]() {
395             std::vector<int> stk;
396             for (int i = l; i <= r; ++i) {
397                 while (stk.size() >= 2 &&
398                     fsign(Point::crossProd(p[id[stk.end()[-2]],
399                         p[id[stk.end()[-1]], p[id[i]]]) < 0)
400                     )
401                     stk.push_back(i);
402             }
403             for (int i = l; i < stk.size(); ++i)
404                 if (stk[i - 1] <= mid && stk[i] > mid)
405                     return std::make_pair(id[stk[i - 1]], id[stk[i]]);
406         };
407
408         auto [ld, rd] = get_base_LR_edge();
409
410         while (true) {
411             addedge(ld, rd);
412             Point ptL = p[ld], ptR = p[rd];
413             int ch = -1, side = -1;
414             for (auto it = neighbor[ld].begin(); it != neighbor[ld].end(); ++it) {
415                 if (fsign(Point::crossProd(ptL, ptR, p[it->v])) > 0 &&
416                     (!-ch || Circle::side(ptL, ptR, p[ch], p[it->v]) < 0)) {
417                     ch = it->v;
418                     side = 0;
419                 }
420             }
421             for (auto it = neighbor[rd].begin(); it != neighbor[rd].end(); ++it) {
422                 if (fsign(Point::crossProd(ptR, p[it->v], ptL)) > 0 &&
423                     (!-ch || Circle::side(ptL, ptR, p[ch], p[it->v]) < 0)) {
424                     ch = it->v;
425                     side = 1;
426                 }
427             }
428             if (!-ch) break;
429             assert(side == 0 || side == 1);
430             if (!side) {
431                 for (auto it = neighbor[ld].begin(); it != neighbor[ld].end(); ++it) {
432                     if (Segment(ptL, p[it->v]).hasIntersection(Segment(ptR, p[ch])) {
433                         neighbor[ld].erase(it);
434                         neighbor[ld].erase(it++);
435                     } else {
436                         ++it;
437                     }
438                 }
439                 ld = ch;
440             } else {
441                 for (auto it = neighbor[rd].begin(); it != neighbor[rd].end(); ++it) {
442                     if (Segment(ptR, p[it->v]).hasIntersection(Segment(ptL, p[ch])) {
443                         neighbor[rd].erase(it);
444                         neighbor[rd].erase(it++);
445                     } else {
446                         ++it;
447                     }
448                 }
449                 rd = ch;
450             }
451         }
452     };
453
454     divide(0, p.size() - 1);
455
456     std::vector<std::pair<int, int>> edges;
457     for (int u = 0; u < p.size(); ++u)
458         for (auto e : neighbor[u])
459             if (u < e.v) edges.emplace_back(u, e.v);
460     return edges;
461 }
462
463 // should be guaranteed that p is strictly convex
464 static std::vector<std::pair<int, int>> furthest(
465     const std::vector<Point>& p) {
466     assert(Polygon::isConvexInCCW(p));
467     std::vector<std::pair<int, int>> edges;
468     if (p.size() < 3) {
469         for (int i = 0; i < p.size(); ++i)
470             for (int j = 0; j < i; ++j) {

```

```

471     edges.emplace_back(i, j);
472 }
473 return edges;
474 }
475
476 std::vector<std::list<Edge>> neighbor(p.size());
477 std::vector<int> ids(p.size());
478 std::iota(ids.begin(), ids.end(), 0);
479
480 // calculate cw, ccw
481 std::vector<int> cw(p.size()), ccw(p.size());
482 for (int i = 0; i < p.size(); ++i) {
483     cw[i] = (i + p.size() - 1) % p.size();
484     ccw[i] = (i + 1) % p.size();
485 }
486 std::random_shuffle(ids.begin(), ids.end());
487 for (int i = ids.size() - 1; i >= 2; --i) {
488     int u = ids[i];
489     std::tie(ccw[cw[u]], cw[ccw[u]]) = std::make_pair(ccw[u], cw[u]);
490 }
491
492 std::vector<std::list<Edge>> lines(p.size());
493 auto bind_rev_edge = [&](std::list<Edge>::iterator lhs,
494     std::list<Edge>::iterator rhs) {
495     lhs->rev = rhs;
496     rhs->rev = lhs;
497 };
498
499 bind_rev_edge(neighbor[ids[0]].emplace(neighbor[ids[0]].begin(), ids[1]),
500     neighbor[ids[1]].emplace(neighbor[ids[1]].begin(), ids[0]));
501
502 for (int i = 2; i < ids.size(); ++i) {
503     int u = ids[i];
504     int cur = ccw[u];
505     auto cur_iter = neighbor[cur].begin();
506     while (1) {
507         while (cur_iter != neighbor[cur].end()) {
508             auto next_iter = std::next(cur_iter);
509             if (next_iter == neighbor[cur].end()) {
510                 if (cur != cw[u]) break;
511             } else {
512                 if (Circle::side(p[cur], p[cur_iter->v], p[next_iter->v], p[u]) < 0)
513                     break;
514             }
515             neighbor[cur].erase(cur_iter++);
516         }
517         bind_rev_edge(neighbor[u].emplace(neighbor[u].begin(), cur),
518             neighbor[cur].emplace(cur_iter, u));
519         if (cur == cw[u]) break;
520         std::tie(cur, cur_iter) =
521             std::make_pair(cur_iter->v, std::next(cur_iter->rev));
522     }
523 }
524
525 for (int u = 0; u < p.size(); ++u)
526     for (auto e : neighbor[u])
527         if (u < e.v) edges.emplace_back(u, e.v);
528 return edges;
529 }
530 };
531
532 struct PlanarGraphDuality {
533     struct DirectionalEdge {
534         int v, id = -1;
535         Point direction;
536         int rev;
537         DirectionalEdge(int v, Point direction) : v(v), direction(direction) {}
538     };
539
540     // return all points' id in each faces and the edges between faces
541     static std::pair<std::vector<std::vector<int>>,
542         std::vector<std::pair<int, int>>>
543     solve(const std::vector<Point>& p,
544         const std::vector<std::pair<int, int>>& edges) {
545         std::vector<DirectionalEdge> directional_edges;
546         directional_edges.reserve(edges.size() * 2);
547         std::vector<std::vector<int>> out_edges(p.size());
548         for (auto [u, v] : edges) {
549             out_edges[u].push_back(directional_edges.size());
550             directional_edges.emplace_back(v, p[v] - p[u]);
551             out_edges[v].push_back(directional_edges.size());
552             directional_edges.emplace_back(u, p[u] - p[v]);
553             directional_edges.end()[-1].rev = out_edges[u].back();
554             directional_edges.end()[-2].rev = out_edges[v].back();
555         }
556         const auto comp = [&, t_comp = Point::PolarComparer()](int lhs, int rhs) {
557             return t_comp(directional_edges[lhs].direction,
558                 directional_edges[rhs].direction);
559         };
560         for (int u = 0; u < p.size(); ++u)
561             std::sort(out_edges[u].begin(), out_edges[u].end(), comp);
562         std::vector<std::vector<int>> faces;
563         for (int u = 0; u < p.size(); ++u) {
564             for (int e_id : out_edges[u]) {
565                 if (~directional_edges[e_id].id) continue;
566                 std::vector<int> pids;
567                 for (int cur_e_id = e_id;;) {
568                     if (~directional_edges[cur_e_id].id) break;
569                     directional_edges[cur_e_id].id = faces.size();
570                     int v = directional_edges[cur_e_id].v;
571                     pids.push_back(v);
572                     auto it = std::lower_bound(out_edges[v].begin(), out_edges[v].end(),
573                         directional_edges[cur_e_id].rev, comp);
574                     assert(*it == directional_edges[cur_e_id].rev);

```

```

575         if (it == out_edges[v].begin())
576             cur_e_id = out_edges[v].back();
577         else
578             cur_e_id = *std::prev(it);
579     }
580     faces.push_back(pids);
581 }
582 }
583 std::vector<std::pair<int, int>> face_edges;
584 for (int u = 0; u < p.size(); ++u)
585     for (int e_id : out_edges[u]) {
586         int rev_id = directional_edges[e_id].rev;
587         if (e_id < rev_id) continue;
588         face_edges.emplace_back(directional_edges[e_id].id,
589                                 directional_edges[rev_id].id);
590     }
591     return std::make_pair(faces, face_edges);
592 }
593 };
594
595 struct Voronoi {
596     static constexpr long double kBoundaryInf = 50000;
597
598     // should be guaranteed that
599     // 1. all points are pairwise distinct
600     // 2. boundary had better to be a convex
601     // 3. all points are inside boundary
602     static std::vector<std::vector<Point>> nearest(
603         const std::vector<Point>& p,
604         const std::vector<Line>& boundary = {
605             Line(Point(-kBoundaryInf, -kBoundaryInf), Point(1, 0)),
606             Line(Point(kBoundaryInf, -kBoundaryInf), Point(0, 1)),
607             Line(Point(kBoundaryInf, kBoundaryInf), Point(-1, 0)),
608             Line(Point(-kBoundaryInf, kBoundaryInf), Point(0, -1)),
609         }) {
610         // p0 in the left
611         auto bisector = [&](const Point& p0, const Point& p1) {
612             auto dir = (p1 - p0).rotate90();
613             auto mid = (p0 + p1) / 2;
614             return Line(mid, dir);
615         };
616         auto edges = Triangulation::nearest(p);
617         std::vector<std::vector<Line>> limit(p.size(), boundary);
618         for (auto [i, j] : edges) {
619             limit[i].push_back(bisector(p[i], p[j]));
620             limit[j].push_back(bisector(p[j], p[i]));
621         }
622         std::vector<std::vector<Point>> regions(p.size());
623         for (int i = 0; i < p.size(); ++i) {
624             regions[i] = HalfPlaneIntersection::solve(limit[i]);
625         }
626         return regions;

```

```

627     }
628
629     // should be guaranteed that
630     // 1. p is strictly convex
631     // 2. boundary had better to be a convex
632     // 3. all points are inside boundary
633     static std::vector<std::vector<Point>> furthest(
634         const std::vector<Point>& p,
635         const std::vector<Line>& boundary = {
636             Line(Point(-kBoundaryInf, -kBoundaryInf), Point(1, 0)),
637             Line(Point(kBoundaryInf, -kBoundaryInf), Point(0, 1)),
638             Line(Point(kBoundaryInf, kBoundaryInf), Point(-1, 0)),
639             Line(Point(-kBoundaryInf, kBoundaryInf), Point(0, -1)),
640         }) {
641         // p0 in the right
642         auto bisector = [&](const Point& p0, const Point& p1) {
643             auto dir = (p0 - p1).rotate90();
644             auto mid = (p0 + p1) / 2;
645             return Line(mid, dir);
646         };
647         auto edges = Triangulation::furthest(p);
648         std::vector<std::vector<Line>> limit(p.size(), boundary);
649         for (auto [i, j] : edges) {
650             limit[i].push_back(bisector(p[i], p[j]));
651             limit[j].push_back(bisector(p[j], p[i]));
652         }
653         std::vector<std::vector<Point>> regions(p.size());
654         for (int i = 0; i < p.size(); ++i) {
655             regions[i] = HalfPlaneIntersection::solve(limit[i]);
656         }
657         return regions;
658     }
659 };
660
661 } // namespace geometry2d

```

***** 圆的面积并 *****

```

1 // Area[i] 表示覆盖次数大于等于i的面积, 复杂度  $O(n^2 \log n)$ 
2 struct P {
3     double x, y;
4     operator+(double _x, double _y) { x = _x, y = _y; }
5     operator+(const P& b) const { return P(x + b.x, y + b.y); }
6     operator-(const P& b) const { return P(x - b.x, y - b.y); }
7     operator*(double b) const { return P(x * b, y * b); }
8     operator/(double b) const { return P(x / b, y / b); }
9     double det(const P& b) const { return x * b.y - y * b.x; }
10    P rot90() const { return P(-y, x); }
11    P unit() { return *this / abs(); }
12    double abs() { return hypot(x, y); }
13 };
14 struct Circle {

```

```

15 P o;
16 double r;
17 bool contain(const Circle& v, const int& c) const { return sgn(r - (o - v.o).abs() - v.r) > c; }
18 bool disjunct(const Circle& v, const int& c) const { // 0严格, -1不严格
19     return sgn((o - v.o).abs() - r - v.r) > c;
20 }
21 };
22 //求圆与圆的交点, 包含相切, 假设无重圆
23 bool isCC(Circle a, Circle b, P& p1, P& p2) {
24     if (a.contain(b, 0) || b.contain(a, 0) || a.disjunct(b, 0)) return 0;
25     double s1 = (a.o - b.o).abs();
26     double s2 = (a.r * a.r - b.r * b.r) / s1;
27     double aa = (s1 + s2) / 2, bb = (s1 - s2) / 2;
28     P mm = (b.o - a.o) * (aa / (aa + bb)) + a.o;
29     double h = sqrt(max(0.0, a.r * a.r - aa * aa));
30     P vv = (b.o - a.o).unit().rot90() * h;
31     p1 = mm + vv, p2 = mm - vv;
32     return 1;
33 }
34 struct EV {
35     P p;
36     double ang;
37     int add;
38     EV() {}
39     EV(const P& _p, double _ang, int _add) { p = _p, ang = _ang, add = _add; }
40     bool operator<(const EV& a) const { return ang < a.ang; }
41 } eve[N * 2];
42 int E, cnt, C, i, j;
43 Circle c[N];
44 bool g[N][N], overlap[N][N];
45 double Area[N];
46 int cX[N], cY[N], cR[N];
47 bool contain(int i, int j) {
48     return (sgn(c[i].r - c[j].r) > 0 || sgn(c[i].r - c[j].r) == 0 && i < j) && c[i].contain(c[j], -1);
49 }
50 int main() {
51     scanf("%d", &C);
52     for (i = 0; i < C; i++) {
53         scanf("%d %d %d", &cX[i], &cY[i], &cR[i]);
54         c[i].o = P(cX[i], cY[i]);
55         c[i].r = cR[i];
56     }
57     for (i = 0; i <= C; i++) Area[i] = 0;
58     for (i = 0; i < C; i++)
59         for (j = 0; j < C; j++) overlap[i][j] = contain(i, j);
60     for (i = 0; i < C; i++)
61         for (j = 0; j < C; j++) g[i][j] = !(overlap[i][j] || overlap[j][i] || c[i].disjunct(c[j], -1));
62     for (i = 0; i < C; i++) {
63         E = 0;
64         cnt = 1;
65         for (j = 0; j < C; j++)
66             if (j != i && overlap[j][i]) cnt++;

```

```

67     for (j = 0; j < C; j++)
68         if (i != j && g[i][j]) {
69             P aa, bb;
70             isCC(c[i], c[j], aa, bb);
71             double A = atan2(aa.y - c[i].o.y, aa.x - c[i].o.x);
72             double B = atan2(bb.y - c[i].o.y, bb.x - c[i].o.x);
73             eve[E++] = EV(bb, B, 1);
74             eve[E++] = EV(aa, A, -1);
75             if (B > A) cnt++;
76         }
77     if (E == 0)
78         Area[cnt] += PI * c[i].r * c[i].r;
79     else {
80         sort(eve, eve + E);
81         eve[E] = eve[0];
82         for (j = 0; j < E; j++) {
83             cnt += eve[j].add;
84             Area[cnt] += eve[j].p.det(eve[j + 1].p) * 0.5;
85             double theta = eve[j + 1].ang - eve[j].ang;
86             if (theta < 0) theta += PI * 2;
87             Area[cnt] += theta * c[i].r * c[i].r * 0.5 - sin(theta) * c[i].r * c[i].r * 0.5;
88         }
89     }
90 }
91 for (i = 1; i <= C; i++) printf("%d %d %.3f\n", i, Area[i] - Area[i + 1]);
92 }

```

4 Math and Number Theory

```

1 //***** combinatorics mod (exLucas) *****
2
3 LL p[maxp], p0, pk[maxp], num[maxp], fac[maxp];
4 void Prime(LL P) { //  $P = p[1]^{num[1]} * \dots * p[p0]^{num[p0]} = pk[1] * \dots * pk[p0]$ 
5     void Pre() {
6         fo(j, 1, p0) {
7             fac[j] = 1;
8             fo(i, 1, pk[j] - 1) if (i % p[j]) fac[j] = fac[j] * i % pk[j];
9         }
10    }
11
12    LL mo;
13    LL Pow(LL x, LL y) { // mod mo
14        LL count(LL n, LL p) { return (n) ? (n/p + count(n/p, p)) : 0; }
15        LL Fc(LL n, LL j) {
16            if (!n) return 1;
17            LL re = Fc(n/p[j], j) * mi(fac[j], n/mo) % mo;
18            fo(i, 1, n % mo) if (i % p[j]) re = re * i % mo;
19            return re;

```

```

20 }
21
22 LL C(LL n, LL m, LL p) { // compute  $C(n+m, n)$ .  $M=n+m$ .
23 // preprocess: Prime(P); Pre();
24 LL ans=0;
25 fo(j, 1, p0) {
26     LL nump=count(M, p[j])-count(m, p[j])-count(n, p[j]);
27     LL phi=pk[j]-pk[j]/p[j];
28     mo=pk[j];
29     LL a=(nump>num[j]) ? 0 : Fc(M, j)*Pow(Fc(n, j), phi-1)%mo*Pow(Fc(m, j), phi-1)%mo*Pow(p[j], nump)%mo
30     (ans+=a *(P/pk[j])%P *Pow(P/pk[j], phi-1)%P)%=P;
31 }
32
33 return ans;
34 }
35
36 //***** Miller_Rabin *****
37
38 int pr[9]={2,3,5,7,11,13,17,19,23};
39 LL mul(LL x, LL y, LL mo) {
40     LL re=0;
41     for(; y; y>>=1, x=(x+x)%mo) if (y&1) re=(re+x)%mo;
42     return re;
43 }
44 LL Pow(LL x, LL y, LL mo) {
45     LL re=1;
46     for(; y; y>>=1, x=mul(x, x, mo)) if (y&1) re=mul(re, x, mo);
47     return re;
48 }
49 bool Miller_Rabin(int d, LL s, LL a, LL n) {
50     a=Pow(a, s, n);
51     if (a==1) return 1;
52     fo(i, 1, d) {
53         if (a==n-1) return 1;
54         if (a==1) return 0;
55         a=mul(a, a, n);
56     }
57     return 0;
58 }
59 bool isprime(LL n) {
60     if (n<2) return 0;
61     fo(i, 0, 8) {
62         if (n==pr[i]) return 1;
63         if (n%pr[i]==0) return 0;
64     }
65     int d=0; LL s=n-1;
66     for(; !(s&1); s>>=1, d++);
67     fo(i, 0, 8) if (!Miller_Rabin(d, s, pr[i], n)) return 0;
68     return 1;
69 }
70
71 //***** pollard_rho *****

```

```

72
73 inline LL ran_f(LL x, LL c, LL n) {return (mul(x, x, n)+c)%n;}
74 LL pollard_rho(LL n) {
75     for(LL c=rand()*rand()%n; ; c=rand()*rand()%n) {
76         LL x=rand()*rand()%n, y=x;
77         for(LL i=0, k=1; ; i++) {
78             x=ran_f(x, c, n);
79             LL t=__gcd(abs(x-y), n);
80             if (t==n) break;
81             else if (t>1) return t;
82             if (i==k) y=x, k<<=1;
83         }
84     }
85 }
86
87 //***** similar gcd *****
88
89 struct FGH{
90     LL f, g, h; //  $f=\sum_{i=0}^n (a*i+b)/c$ ,  $g=\sum_{i=0}^n i*(a*i+b)/c$ ,  $h=\sum_{i=0}^n ((a*i+b)/c)^2$ 
91 };
92
93 FGH calc(LL a, LL b, LL c, LL n) {
94     LL ac=a/c, bc=b/c, sum1=n*(n+1)%mo*inv2%mo, sum2=n*(n+1)%mo*(2*n+1)%mo*inv6%mo;
95     if (!a) return (FGH){(n+1)*bc%mo, sum1*bc%mo, (n+1)*bc%mo*bc%mo};
96     if (a>c || b>c) {
97         FGH nxt=calc(a%c, b%c, c, n);
98         LL f=(nxt.f+ac*sum1+(n+1)*bc)%mo;
99         LL g=(nxt.g+ac*sum2+sum1*bc)%mo;
100         LL h=(nxt.h+sum2*ac%mo*ac%mo+(n+1)*bc%mo*bc%mo+2*ac*nxt.g%mo+2*bc*nxt.f%mo+n*(n+1)%mo*ac%mo*bc%mo)%mo;
101         return (FGH){f, g, h};
102     } else {
103         LL m=(a*n+b)/c;
104         FGH nxt=calc(c, c-b-1, a, m-1);
105         m%=mo;
106         LL f=(m*n-nxt.f+mo)%mo;
107         LL g=((n+1)*n%mo*m-nxt.f-nxt.h+mo)%mo*inv2%mo;
108         LL h=((m+1)*n%mo*m-nxt.g-nxt.g-f-nxt.f-nxt.f+mo*5)%mo;
109         return (FGH){f, g, h};
110     }
111 }
112
113 //***** 筛 *****
114
115 //  $f(n, k)$  表示把  $n$  拆成  $k$  个数的积的方案数
116 LL mw[2*maxsqtrn], g[2*maxsqtrn];
117 int w0, id1[maxsqtrn], id2[maxsqtrn];
118 LL min25_g(LL n) {
119     w0=0;
120     for(LL i=1, j; i<=n; i=j+1) {
121         j=n/(n/i);

```



```

123     mw[++w0]=n/i;
124     if (mw[w0]<=sqrtn) id1[mw[w0]]=w0; else id2[j]=w0;
125     g[w0]=mw[w0]-1;
126 }
127 fo(j,1,Np[sqrtn])
128     for(int i=1; i<=w0 && (LL)p[j]*p[j]<=mw[i]; i++) {
129         int id=(mw[i]/p[j]<=sqrtn) ?id1[mw[i]/p[j]] :id2[n/(mw[i]/p[j])];
130         (g[i]-=g[id]-(j-1))%=mo;
131     }
132 }
133 LL min25_S(LL x,int j,int k) {
134     if (x<=1 || p[j]>x) return 0;
135     int id=(x<=sqrtn) ?id1[x] :id2[n/x];
136     LL re=(g[id]-(j-1))*k;
137     for(int i=j; i<=Np[sqrtn] && (LL)p[i]*p[i]<=x; i++) {
138         LL pe=p[i];
139         for(int e=1; pe*p[i]<=x; e++, pe*=p[i])
140             (re+=min25_S(x/pe,i+1,k)*C[e+k-1][k-1]+C[e+k][k-1])%=mo;
141     }
142     return re;
143 }

```

5 Others

```

1  //***** dominator tree (other's) *****
2
3  const int maxn = 310000;
4  const int maxm = 1050000;
5
6  int n,m,s;
7  int sdom[maxn],idom[maxn];
8  vector<int>V[maxn];
9  vector<int>g[maxn],e[maxn];
10
11 int dfn[maxn],To[maxn],id,par[maxn];
12 void build(int u){
13     To[dfn[u]=++id]=u;
14     for (auto v:g[u]) if (!dfn[v]) par[v]=u,build(v);
15 }
16
17 int fa[maxn],fas[maxn];
18 void find(int x)
19 {
20     if(fa[x]==x) return;
21     find(fa[x]);
22     if(dfn[sdom[fas[fa[x]]]]<dfn[sdom[fas[x]]]) fas[x]=fas[fa[x]];
23     fa[x]=fa[fa[x]];
24 }

```

```

25 int ans[maxn];
26
27 int main(){
28     scanf("%d%d%d",&n,&m,&s);
29     int nn=n;
30     rep(i,1,m) {
31         int x,y;
32         scanf("%d%d",&x,&y);
33         e[n+i].push_back(x); e[y].push_back(n+i);
34         g[x].push_back(n+i); g[n+i].push_back(y);
35     }
36     n+=m;
37     build(s);
38     rep(i,1,n) fa[i]=i,fas[i]=i,sdom[i]=idom[i]=i;
39     per(i,id,1){
40         int x=To[i],&semi=sdom[x];
41         for(auto y:e[x]) if (dfn[y]){
42             find(y);
43             if(dfn[semi]>dfn[sdom[fas[y]]]) semi=sdom[fas[y]];
44         }
45         for(auto y:V[x]){
46             find(y);
47             if(dfn[sdom[fas[y]]]<i) idom[y]=fas[y];
48             else idom[y]=x;
49         }
50         V[semi].push_back(x);
51         for (auto y:g[x]) if (par[y]==x) fa[y]=x;
52     }
53     rep(i,1,id){
54         int x=To[i];
55         if(idom[x]!=sdom[x]) idom[x]=idom[idom[x]];
56     }
57     per(i,id,2){
58         int x=To[i];
59         if (1<=x&&x<=n&&idom[x]>nn) ans[idom[x]-nn]=1;
60     }
61     int cnt=0;
62     rep(i,1,m) if (!ans[i]) cnt++;
63     printf("%d\n",cnt);
64     rep(i,1,m) if (!ans[i]) printf("%d ",i);
65     return 0;
66 }
67
68 //***** dq_mincut *****
69
70 int num,st[maxn];
71 bool vis[maxn];
72 void find(int k)
73 {
74     vis[k]=1;
75     st[++num]=k;
76     for(int p=f1[k]; p; p=nxt[p]) if (!vis[go[p]]) find(go[p]);

```

```

77 }
78
79 int bj[maxn],sum,nowh[maxn],d[maxn],bz[maxn],bzcnt;
80 // maxflow here, 稀疏图 dinic, else isap
81 void bfs(int s)
82 {
83     bz[ d[1]=s ]+=bzcnt;
84     for(int i=1, j=1; i<=j; i++)
85     {
86         for(int p=f1[d[i]]; p; p=nxt[p]) if (val2[p] && bz[go[p]]!=bzcnt)
87             bz[ d[++j]=go[p] ]=bzcnt;
88     }
89 }
90
91 int st1[maxn];
92 vector<pair<int,int>> e[maxn];
93 void Mincut(int l,int r)
94 {
95     memcpy(val2,val,sizeof(val));
96     sum=st[r];
97
98     int flow=0;
99     while (Dinic_bfs(st[l])) flow+=Dinic_dfs(st[l],inf);
100    e[st[l]].push_back(make_pair(sum,flow)), e[sum].push_back(make_pair(st[l],flow));
101
102    bfs(st[l]);
103    int newr=l-1, newl=r+1;
104    fo(i,l,r) if (bz[st[i]]==bzcnt) st1[++newr]=st[i]; else st1[--newl]=st[i];
105    fo(i,l,r) st[i]=st1[i];
106
107    if (l<newr) Mincut(l,newr);
108    if (newl<r) Mincut(newl,r);
109 }
110
111 int main()
112 {
113     fo(i,1,n) if (!vis[i] && f1[i])
114     {
115         num=0;
116         find(i);
117         Mincut(1,num);
118     }
119 }
120
121 //***** KM *****
122
123 LL lx[maxn],ly[maxn],slack[maxn];
124 int f[maxn],pre[maxn];
125 bool vis[maxn];
126 LL KM(int nl,int nr)
127 {
128     fo(i,1,nl)

```

```

129         fo(j,1,nr) lx[i]=max(lx[i],mp[i][j]);
130     fo(i,1,nl)
131     {
132         memset(slack,127,sizeof(LL)*(nr+1));
133         memset(vis,0,sizeof(bool)*(nr+1));
134         f[0]=i;
135         int py=0, nextpy;
136         for(; f[py]; py=nextpy)
137         {
138             int px=f[py];
139             LL d=inf;
140             vis[py]=1;
141             fo(j,1,nr) if (!vis[j])
142             {
143                 if (lx[px]+ly[j]-mp[px][j]<slack[j]) slack[j]=lx[px]+ly[j]-mp[px][j], pre[j]=py;
144                 if (slack[j]<d) d=slack[j], nextpy=j;
145             }
146             fo(j,0,nr) if (vis[j]) lx[f[j]]-=d, ly[j]+=d;
147             else slack[j]-=d;
148         }
149         for(; py; py=pre[py]) f[py]=f[pre[py]];
150     }
151     LL re=0;
152     fo(i,1,nl) re+=lx[i];
153     fo(j,1,nr) re+=ly[j];
154     return re;
155 }
156
157 //***** LCA *****
158
159 //倍增
160 int deep[maxn],fa[maxn][MX+1];
161 int lca(int x,int y) {
162     if (deep[x]<deep[y]) swap(x,y);
163     fd(i,MX,0)
164         while (deep[fa[x][i]]>=deep[y]) x=fa[x][i];
165     if (x==y) return x;
166     fd(i,MX,0)
167         while (fa[x][i]!=fa[y][i]) x=fa[x][i], y=fa[y][i];
168     return fa[x][0];
169 }
170
171 //tarjan
172 int totq,goq[2*maxm],num[2*maxm],nextq[2*maxm],fq[maxn];
173 void inq(int x,int y,int z) {
174     goq[++totq]=y;
175     num[totq]=z;
176     nextq[totq]=fq[x];
177     fq[x]=totq;
178 }
179 int lca[maxm],fa[maxn];
180 bool bz[maxn];

```

```

181 int get(int x) {
182     if (fa[x]==x) return x;
183     return fa[x]=get(fa[x]);
184 }
185 void tarjan(int k,int last) { //ordinary
186     fa[k]=k;
187     for(int p=f1[k]; p; p=next[p]) if (go[p]!=last) {
188         tarjan(go[p],k);
189         fa[go[p]]=k;
190     }
191     bz[k]=1;
192     for(int p=fq[k]; p; p=nextq[p])
193         if (bz[goq[p]]) lca[num[p]]=get(goq[p]); else inq(goq[p],k,num[p]);
194 }
195
196 void tarjan(int k,int last) { //支持维护值
197     fa[k]=k;
198     for(int p=f1[k]; p; p=next[p]) if (go[p]!=last) {
199         tarjan(go[p],k);
200         f[go[p]]+=val[p];
201         fa[go[p]]=k;
202     }
203     bz[k]=1;
204     for(int p=fq[k]; p; p=nextq[p]) if (bz[goq[p]]) {
205         int t=get(goq[p]);
206         ans[num[p]]=valq[p]+f[goq[p]];
207         if (t!=k) inq(t,k,f[goq[p]],num[p]); else lca[num[p]]=t;
208     } else inq(goq[p],k,0,num[p]);
209 }
210
211 //rmq
212 int fa[2*maxn][MX+5],deep[maxn],ap[2*maxn],fir[2*maxn],Log[2*maxn],er[MX+5];
213 void rmq_pre() {
214     fo(i,1,ap[0]) fa[i][0]=ap[i], Log[i]=log(i)/log(2);
215     fo(i,0,MX) er[i]=1<<i;
216     fo(j,1,MX)
217         fo(i,1,ap[0]) {
218             fa[i][j]=fa[i][j-1];
219             if (i+er[j-1]<=ap[0] && deep[fa[i+er[j-1]][j-1]]<deep[fa[i][j]])
220                 fa[i][j]=fa[i+er[j-1]][j-1];
221         }
222 }
223 int lca(int x,int y) {
224     x=fir[x], y=fir[y];
225     if (x>y) swap(x,y);
226     int t=Log[y-x+1];
227     return (deep[fa[x][t]]<deep[fa[y-er[t]+1][t]]) ?fa[x][t] :fa[y-er[t]+1][t] ;
228 }
229
230 void dfs_pre(int k,int last) {
231     deep[k]=deep[last]+1;
232     ap[++ap[0]]=k, fir[k]=ap[0];

```

```

233     for(int p=f1[k]; p; p=next[p]) if (go[p]!=last) {
234         dfs_pre(go[p],k);
235         ap[++ap[0]]=k;
236     }
237 }
238
239 //***** LeftTree *****
240
241 struct node{
242     int val,l,r,fa,dis;
243 };
244
245 node lt[maxn];
246 int tot,ga[maxn];
247 int New(int val=0)
248 {
249     lt[++tot]=(node){val,0,0,0,0};
250     ga[tot]=tot;
251     return tot;
252 }
253 int merge(int a,int b)
254 {
255     if (!a) return b;
256     if (!b) return a;
257     if (lt[a].val>lt[b].val || lt[a].val==lt[b].val && a>b) swap(a,b);
258     lt[a].r=merge(lt[a].r,b);
259     lt[lt[a].r].fa=a;
260     ga[lt[a].r]=a;
261     if (lt[lt[a].r].dis>lt[lt[a].l].dis) swap(lt[a].l,lt[a].r);
262     lt[a].dis=(lt[a].r==0) ?0 :lt[lt[a].r].dis+1;
263     return a;
264 }
265 int top(int x) {return (ga[x]==x) ?x :ga[x]=top(ga[x]) ;}
266 void pop(int x)
267 {
268     int t=merge(lt[x].l,lt[x].r);
269     lt[t].fa=lt[x].fa;
270     ga[x]=(top(x)==x) ?t :top(x);
271     ga[t]=ga[x];
272     for(int i=lt[x].fa; i; i=lt[i].fa) if (lt[lt[i].l].dis<lt[lt[i].r].dis)
273     {
274         swap(lt[i].l,lt[i].r);
275         lt[i].dis=lt[lt[i].r].dis+1;
276     } else break;
277     lt[x].fa=-1;
278 }
279 void push(int x,int val)
280 {
281     merge(top(x),New(val));
282 }
283
284 // init : lt[0].dis=-1;

```

```

285
286 //***** maxflow *****
287
288 //isap+gap+当前弧
289 int bj[maxsum],gap[maxsum],sum,nowh[maxsum],d[maxsum];
290 void init_Maxflow() {
291     memset(bj,0,sizeof(bj));
292     memset(gap,0,sizeof(gap)); gap[0]=sum+1;
293     memcpy(nowh,f1,sizeof(nowh));
294     d[1]=sum;
295     for(int i=1, j=1; i<=j; i++) {
296         for(int p=f1[d[i]]; p; p=e[p].nxt) if (e[p].go!=sum && !bj[e[p].go]) {
297             bj[e[p].go]=bj[d[i]]+1;
298             gap[0]--, gap[bj[e[p].go]]++;
299             d[++j]=e[p].go;
300         }
301     }
302 }
303 int Maxflow(int k,int flow) {
304     if (k==sum) return flow;
305     int re=0;
306     for(int &p=nowh[k]; p; p=e[p].nxt) if (e[p].val && bj[k]==bj[e[p].go]+1) {
307         int fl=Maxflow(e[p].go, (flow-re<e[p].val) ?(flow-re) :e[p].val);
308         e[p].val-=fl;
309         e[(p&1) ?p+1 :p-1 ].val+=fl;
310         re+=fl;
311         if (re==flow || bj[0]>sum) return re;
312     }
313     nowh[k]=f1[k];
314     if ((--gap[bj[k]])==0) bj[0]=sum+1; else bj[k]++;
315     gap[bj[k]]++;
316     return re;
317 }
318 //dinic+当前弧
319 bool Dinic_bfs(int s) {
320     memset(bj,255,sizeof(bj));
321     memcpy(nowh,f1,sizeof(nowh));
322     bj[ d[1]=s ]=0;
323     for(int i=1, j=1; i<=j; i++) {
324         for(int p=f1[d[i]]; p; p=e[p].nxt) if (e[p].val && bj[e[p].go]==-1) {
325             bj[e[p].go]=bj[d[i]]+1;
326             d[++j]=e[p].go;
327         }
328     }
329     return bj[sum]!=-1;
330 }
331 int Dinic_dfs(int k,int flow) {
332     if (k==sum) return flow;
333     int re=0;
334     for(int &p=nowh[k]; p; p=e[p].nxt) if (e[p].val && bj[k]+1==bj[e[p].go]) {
335         int fl=Dinic_dfs(e[p].go, (flow-re<e[p].val) ?(flow-re) :e[p].val);
336         e[p].val-=fl;

```

```

337         e[(p&1) ?p+1 :p-1 ].val+=fl;
338         re+=fl;
339         if (re==flow) return re;
340     }
341     return re;
342 }
343
344 //***** random Hung *****
345
346 int bz[maxn],tim,pt[maxn];
347 bool Hung(int x,int tim) {
348     if (bz[x]==tim) return 0;
349     bz[x]=tim;
350     random_shuffle(e[x].begin(),e[x].end());
351     for(int go:e[x]) {
352         int k=pt[go];
353         pt[k]=0, pt[x]=go, pt[go]=x;
354         if (!k || Hung(k,tim)) return 1;
355         pt[k]=go, pt[go]=k, pt[x]=0;
356     }
357     return 0;
358 }
359 int main() {
360     fo(i,1,n) pmt[i]=i;
361     random_shuffle(pmt+1,pmt+1+n);
362     int tim=0, cnt=0;
363     fo(j,1,5)
364         fo(i,1,n) if (!pt[pmt[i]]) Hung(pmt[i],++tim);
365 }
366
367 //***** round square tree *****
368
369 //广义 (任意路径都是圆方交替)
370 int sum,dfn[maxn],low[maxn],z[maxn],z0,num[2*maxn],nn;
371 vector<int> e[2*maxn];
372 void tarjan(int k,int last) {
373     dfn[k]=low[k]=++sum;
374     z[++z0]=k;
375     for(int p=f1[k]; p; p=nxt[p]) if (bh[p]!=last) {
376         if (!dfn[go[p]]) {
377             tarjan(go[p],bh[p]);
378             low[k]=min(low[k],low[go[p]]);
379
380             if (low[go[p]]>=dfn[k]) {
381                 num[++nn]=1;
382                 e[nn].push_back(k), e[k].push_back(nn);
383                 do {
384                     num[nn]++;
385                     e[nn].push_back(z[z0]), e[z[z0]].push_back(nn);
386                 } while (z[z0--]!=go[p]);
387             }
388             } else low[k]=min(low[k],dfn[go[p]]);

```

```

389     }
390 }
391
392 //***** virtual tree *****
393
394 int p0,p[2*maxn],z[maxn],z0;
395 bool cmpP(const int &a,const int &b) {return dfn[a]<dfn[b];}
396 void make_vtree() {
397     tot=0;
398     sort(p+1,p+1+p0,cmpP);
399     int t=p0;
400     fo(i,1,t-1) p[++p0]=lca(p[i],p[i+1]);
401     sort(p+1,p+1+p0,cmpP);
402     f1[ z[z0=1]=1 ]=0;
403     p[0]=1;
404     fo(i,1,p0) if (p[i]!=p[i-1]) {
405         for(; z0 && (dfn[p[i]]<dfn[z[z0]] || en[z[z0]]<dfn[p[i]]); z0--) ins(z[z0-1],z[z0]);
406         f1[ z[++z0]=p[i] ]=0;

```

```

407     }
408     fo(i,1,z0-1) ins(z[i],z[i+1]);
409 }
410
411 //***** Tools *****
412
413 //乘法取模黑科技 Claris
414 LL mul(LL a,LL b,LL n){return(a*b-(LL)(a/(long double)n*b+1e-3)*n+n)%n;}
415
416 //split a string by whitespace
417 vector<string> split_str(string str) {
418     vector<string> result;
419     istringstream iss(str);
420     string s;
421     while ( getline( iss, s, ' ' ) ) result.push_back(s);
422     return result;
423 }

```