```
ICPC Library @ tapu
------------------------------------------------
|                      tree                    |
------------------------------------------------

######################################################
############### tree-diameter.cpp ################
######################################################

template< typename T >
pair< T, int > dfs(const WeightedGraph< T > &g, int idx, int par) {
  pair< T, int > ret(0, idx);
  for(auto &e : g[idx]) {
    if(e.to == par) continue;
    auto cost = dfs(g, e.to, idx);
    cost.first += e.cost;
    ret = max(ret, cost);
  }
  return ret;
}

template< typename T >
T tree_diameter(const WeightedGraph< T > &g) {
  auto p = dfs(g, 0, -1);
  auto q = dfs(g, p.second, -1);
  return (q.first);
}


######################################################
###### doubling-lowest-common-ancestor.cpp #######
######################################################

template< typename G >
struct DoublingLowestCommonAncestor {
  const int LOG;
  vector< int > dep;
  const G &g;
  vector< vector< int > > table;

  DoublingLowestCommonAncestor(const G &g) : g(g), dep(g.size()), LOG(32 - __built
in_clz(g.size())) {
    table.assign(LOG, vector< int >(g.size(), -1));
  }

  void dfs(int idx, int par, int d) {
    table[0][idx] = par;
    dep[idx] = d;
    for(auto &to : g[idx]) {
      if(to != par) dfs(to, idx, d + 1);
    }
  }

  void build() {
    dfs(0, -1, 0);
    for(int k = 0; k + 1 < LOG; k++) {
      for(int i = 0; i < table[k].size(); i++) {
        if(table[k][i] == -1) table[k + 1][i] = -1;
        else table[k + 1][i] = table[k][table[k][i]];
      }
    }
  }

  int query(int u, int v) {
    if(dep[u] > dep[v]) swap(u, v);
    for(int i = LOG - 1; i >= 0; i--) {
      if(((dep[v] - dep[u]) >> i) & 1) v = table[i][v];
    }
    if(u == v) return u;
    for(int i = LOG - 1; i >= 0; i--) {
```

```
   70:       if(table[i][u] != table[i][v]) {
   71:         u = table[i][u];
   72:         v = table[i][v];
   73:       }
   74:     }
   75:     return table[0][u];
   76:   }
   77: };
   78:
   79:
   80: ################################################
   81: ############## tree-isomorphism.cpp ##############
   82: ################################################
   83:
   84: template< typename G >
   85: bool tree_isomorphism(const G &a, const G &b) {
   86:   if(a.size() != b.size()) return false;
   87:
   88:   const int N = (int) a.size();
   89:   using pvi = pair< vector< int >, vector< int > >;
   90:
   91:   auto get_uku = [&](const G &t, int e) {
   92:     stack< pair< int, int > > st;
   93:     st.emplace(e, -1);
   94:     vector< int > dep(N, -1), par(N);
   95:     while(!st.empty()) {
   96:       auto p = st.top();
   97:       if(dep[p.first] == -1) {
   98:         dep[p.first] = p.second == -1 ? 0 : dep[p.second] + 1;
   99:         for(auto &to : t[p.first]) if(to != p.second) st.emplace(to, p.first);
  100:       } else {
  101:         par[p.first] = p.second;
  102:         st.pop();
  103:       }
  104:     }
  105:     return make_pair(dep, par);
  106:   };
  107:
  108:   auto solve = [&](const pvi &latte, const pvi &malta) {
  109:
  110:     int d = *max_element(begin(latte.first), end(latte.first));
  111:     if(d != *max_element(begin(malta.first), end(malta.first))) return false;
  112:
  113:     vector< vector< int > > latte_d(d + 1), malta_d(d + 1), latte_key(N), malta_ke
y(N);
  114:
  115:     for(int i = 0; i < N; i++) latte_d[latte.first[i]].emplace_back(i);
  116:     for(int i = 0; i < N; i++) malta_d[malta.first[i]].emplace_back(i);
  117:
  118:     for(int i = d; i >= 0; i--) {
  119:       map< vector< int >, int > ord;
  120:       for(auto &idx : latte_d[i]) {
  121:         sort(begin(latte_key[idx]), end(latte_key[idx]));
  122:         ord[latte_key[idx]]++;
  123:       }
  124:       for(auto &idx : malta_d[i]) {
  125:         sort(begin(malta_key[idx]), end(malta_key[idx]));
  126:         if(--ord[malta_key[idx]] < 0) return false;
  127:       }
  128:       if(i == 0) return ord.size() == 1;
  129:
  130:       int ptr = 0;
  131:       for(auto &p : ord) {
  132:         if(p.second != 0) return false;
  133:         p.second = ptr++;
  134:       }
  135:       for(auto &idx : latte_d[i]) {
  136:         latte_key[latte.second[idx]].emplace_back(ord[latte_key[idx]]);
  137:       }
  138:       for(auto &idx : malta_d[i]) {
```

```
  139:         malta_key[malta.second[idx]].emplace_back(ord[malta_key[idx]]);
  140:       }
  141:     }
  142:     assert(0);
  143:   };
  144:   auto p = centroid(a), q = centroid(b);
  145:   if(p.size() != q.size()) return false;
  146:   auto a1 = get_uku(a, p[0]);
  147:   auto b1 = get_uku(b, q[0]);
  148:   if(solve(a1, b1)) return true;
  149:   if(p.size() == 1) return false;
  150:   auto a2 = get_uku(a, p[1]);
  151:   return solve(a2, b1);
  152: }
  153:
  154:
  155: ##################################################
  156: ######### heavy-light-decomposition.cpp ##########
  157: ##################################################
  158:
  159: template< typename G >
  160: struct HeavyLightDecomposition {
  161:   G &g;
  162:   vector< int > sz, in, out, head, rev, par;
  163:
  164:   HeavyLightDecomposition(G &g) :
  165:       g(g), sz(g.size()), in(g.size()), out(g.size()), head(g.size()), rev(g.size(
)), par(g.size()) {}
  166:
  167:   void dfs_sz(int idx, int p) {
  168:     par[idx] = p;
  169:     sz[idx] = 1;
  170:     if(g[idx].size() && g[idx][0] == p) swap(g[idx][0], g[idx].back());
  171:     for(auto &to : g[idx]) {
  172:       if(to == p) continue;
  173:       dfs_sz(to, idx);
  174:       sz[idx] += sz[to];
  175:       if(sz[g[idx][0]] < sz[to]) swap(g[idx][0], to);
  176:     }
  177:   }
  178:
  179:   void dfs_hld(int idx, int par, int &times) {
  180:     in[idx] = times++;
  181:     rev[in[idx]] = idx;
  182:     for(auto &to : g[idx]) {
  183:       if(to == par) continue;
  184:       head[to] = (g[idx][0] == to ? head[idx] : to);
  185:       dfs_hld(to, idx, times);
  186:     }
  187:     out[idx] = times;
  188:   }
  189:
  190:   void build() {
  191:     dfs_sz(0, -1);
  192:     int t = 0;
  193:     dfs_hld(0, -1, t);
  194:   }
  195:
  196:   int la(int v, int k) {
  197:     while(1) {
  198:       int u = head[v];
  199:       if(in[v] - k >= in[u]) return rev[in[v] - k];
  200:       k -= in[v] - in[u] + 1;
  201:       v = par[u];
  202:     }
  203:   }
  204:
  205:   int lca(int u, int v) {
  206:     for(;; v = par[head[v]]) {
  207:       if(in[u] > in[v]) swap(u, v);
```

```
208:       if(head[u] == head[v]) return u;
209:     }
210:   }
211:
212:   template< typename T, typename Q, typename F >
213:   T query(int u, int v, const T &ti, const Q &q, const F &f, bool edge = false) {
214:     T l = ti, r = ti;
215:     for(;; v = par[head[v]]) {
216:       if(in[u] > in[v]) swap(u, v), swap(l, r);
217:       if(head[u] == head[v]) break;
218:       l = f(q(in[head[v]], in[v] + 1), l);
219:     }
220:     return f(f(q(in[u] + edge, in[v] + 1), l), r);
221: //  return {f(q(in[u] + edge, in[v] + 1), l), r};
222:   }
223:
224:   template< typename Q >
225:   void add(int u, int v, const Q &q, bool edge = false) {
226:     for(;; v = par[head[v]]) {
227:       if(in[u] > in[v]) swap(u, v);
228:       if(head[u] == head[v]) break;
229:       q(in[head[v]], in[v] + 1);
230:     }
231:     q(in[u] + edge, in[v] + 1);
232:   }
233: };
234:
235:
236: ######################################################
237: ################## rerooting.cpp ###################
238: ######################################################
239:
240: template< typename Data, typename T >
241: struct ReRooting {
242:
243:   struct Node {
244:     int to, rev;
245:     Data data;
246:   };
247:
248:   using F1 = function< T(T, T) >;
249:   using F2 = function< T(T, Data) >;
250:
251:   vector< vector< Node > > g;
252:   vector< vector< T > > ldp, rdp;
253:   vector< int > lptr, rptr;
254:   const F1 f1;
255:   const F2 f2;
256:   const T ident;
257:
258:   ReRooting(int n, const F1 &f1, const F2 &f2, const T &ident) :
259:       g(n), ldp(n), rdp(n), lptr(n), rptr(n), f1(f1), f2(f2), ident(ident) {}
260:
261:   void add_edge(int u, int v, const Data &d) {
262:     g[u].emplace_back((Node) {v, (int) g[v].size(), d});
263:     g[v].emplace_back((Node) {u, (int) g[u].size() - 1, d});
264:   }
265:
266:   void add_edge_bi(int u, int v, const Data &d, const Data &e) {
267:     g[u].emplace_back((Node) {v, (int) g[v].size(), d});
268:     g[v].emplace_back((Node) {u, (int) g[u].size() - 1, e});
269:   }
270:
271:
272:   T dfs(int idx, int par) {
273:
274:     while(lptr[idx] != par && lptr[idx] < g[idx].size()) {
275:       auto &e = g[idx][lptr[idx]];
276:       ldp[idx][lptr[idx] + 1] = f1(ldp[idx][lptr[idx]], f2(dfs(e.to, e.rev), e.dat
a));
```

```
  277:       ++lptr[idx];
  278:     }
  279:     while(rptr[idx] != par && rptr[idx] >= 0) {
  280:       auto &e = g[idx][rptr[idx]];
  281:       rdp[idx][rptr[idx]] = f1(rdp[idx][rptr[idx] + 1], f2(dfs(e.to, e.rev), e.dat
a));
  282:       --rptr[idx];
  283:     }
  284:     if(par < 0) return rdp[idx][0];
  285:     return f1(ldp[idx][par], rdp[idx][par + 1]);
  286:   }
  287:
  288:   vector< T > solve() {
  289:     for(int i = 0; i < g.size(); i++) {
  290:       ldp[i].assign(g[i].size() + 1, ident);
  291:       rdp[i].assign(g[i].size() + 1, ident);
  292:       lptr[i] = 0;
  293:       rptr[i] = (int) g[i].size() - 1;
  294:     }
  295:     vector< T > ret;
  296:     for(int i = 0; i < g.size(); i++) {
  297:       ret.push_back(dfs(i, -1));
  298:     }
  299:     return ret;
  300:   }
  301: };
  302:
  303:
  304: ######################################################
  305: ############ convert-rooted-tree.cpp #############
  306: ######################################################
  307:
  308: template< typename G >
  309: G convert_rooted_tree(const G &g, int r = 0) {
  310:   int N = (int) g.size();
  311:   G rg(N);
  312:   vector< int > v(N);
  313:   v[r] = 1;
  314:   queue< int > que;
  315:   que.emplace(r);
  316:   while(!que.empty()) {
  317:     auto p = que.front();
  318:     que.pop();
  319:     for(auto &to : g[p]) {
  320:       if(v[to] == 0) {
  321:         v[to] = 1;
  322:         que.emplace(to);
  323:         rg[p].emplace_back(to);
  324:       }
  325:     }
  326:   }
  327:   return rg;
  328: }
  329:
  330:
  331: ######################################################
  332: ########### centroid-decomposition.cpp ###########
  333: ######################################################
  334:
  335: template< typename G >
  336: struct CentroidDecomposition {
  337:   const G &g;
  338:   vector< int > sub;
  339:   vector< vector< int > > belong;
  340:   vector< bool > v;
  341:
  342:   CentroidDecomposition(const G &g) : g(g), sub(g.size()), v(g.size()), belong(g.s
ize()) {}
  343:
  344:   inline int build_dfs(int idx, int par) {
```

```
345:     sub[idx] = 1;
346:     for(auto &to : g[idx]) {
347:       if(to == par || v[to]) continue;
348:       sub[idx] += build_dfs(to, idx);
349:     }
350:     return sub[idx];
351:   }
352:
353:   inline int search_centroid(int idx, int par, const int mid) {
354:     for(auto &to : g[idx]) {
355:       if(to == par || v[to]) continue;
356:       if(sub[to] > mid) return search_centroid(to, idx, mid);
357:     }
358:     return idx;
359:   }
360:
361:   inline void belong_dfs(int idx, int par, int centroid) {
362:     belong[idx].emplace_back(centroid);
363:     for(auto &to : g[idx]) {
364:       if(to == par || v[to]) continue;
365:       belong_dfs(to, idx, centroid);
366:     }
367:   }
368:
369:   inline int build(UnWeightedGraph &t, int idx) {
370:     int centroid = search_centroid(idx, -1, build_dfs(idx, -1) / 2);
371:     v[centroid] = true;
372:     belong_dfs(centroid, -1, centroid);
373:     for(auto &to : g[centroid]) {
374:       if(!v[to]) t[centroid].emplace_back(build(t, to));
375:     }
376:     v[centroid] = false;
377:     return centroid;
378:   }
379:
380:   inline int build(UnWeightedGraph &t) {
381:     t.resize(g.size());
382:     return build(t, 0);
383:   }
384: };
385:
386:
387: ######################################################
388: ################## centroid.cpp ##################
389: ######################################################
390:
391: template< typename G >
392: vector< int > centroid(const G &g) {
393:   const int N = (int) g.size();
394:
395:   stack< pair< int, int > > st;
396:   st.emplace(0, -1);
397:   vector< int > sz(N), par(N);
398:   while(!st.empty()) {
399:     auto p = st.top();
400:     if(sz[p.first] == 0) {
401:       sz[p.first] = 1;
402:       for(auto &to : g[p.first]) if(to != p.second) st.emplace(to, p.first);
403:     } else {
404:       for(auto &to : g[p.first]) if(to != p.second) sz[p.first] += sz[to];
405:       par[p.first] = p.second;
406:       st.pop();
407:     }
408:   }
409:
410:   vector< int > ret;
411:   int size = N;
412:   for(int i = 0; i < N; i++) {
413:     int val = N - sz[i];
414:     for(auto &to : g[i]) if(to != par[i]) val = max(val, sz[to]);
```

```
415:     if(val < size) size = val, ret.clear();
416:     if(val == size) ret.emplace_back(i);
417:   }
418:
419:   return ret;
420: }
421:
422:
423: ------------------------------------------------
424: |                    graph                     |
425: ------------------------------------------------
426:
427: ################################################
428: ################## boruvka.cpp ##################
429: ################################################
430:
431: template< typename T, typename F >
432: T boruvka(int N, F f) {
433:   vector< int > rev(N), belong(N);
434:   UnionFind uf(N);
435:   T ret = T();
436:   while(uf.size(0) != N) {
437:     int ptr = 0;
438:     for(int i = 0; i < N; i++) {
439:       if(uf.find(i) == i) {
440:         belong[i] = ptr++;
441:         rev[belong[i]] = i;
442:       }
443:     }
444:     for(int i = 0; i < N; i++) {
445:       belong[i] = belong[uf.find(i)];
446:     }
447:     auto v = f(ptr, belong);
448:     bool update = false;
449:     for(int i = 0; i < ptr; i++) {
450:       if(~v[i].second && uf.unite(rev[i], rev[v[i].second])) {
451:         ret += v[i].first;
452:         update = true;
453:       }
454:     }
455:     if(!update) return -1; // notice!!
456:   }
457:   return ret;
458: }
459:
460:
461: ################################################
462: ################## dijkstra.cpp ##################
463: ################################################
464:
465: template< typename T >
466: vector< T > dijkstra(WeightedGraph< T > &g, int s) {
467:   const auto INF = numeric_limits< T >::max();
468:   vector< T > dist(g.size(), INF);
469:
470:   using Pi = pair< T, int >;
471:   priority_queue< Pi, vector< Pi >, greater< Pi > > que;
472:   dist[s] = 0;
473:   que.emplace(dist[s], s);
474:   while(!que.empty()) {
475:     T cost;
476:     int idx;
477:     tie(cost, idx) = que.top();
478:     que.pop();
479:     if(dist[idx] < cost) continue;
480:     for(auto &e : g[idx]) {
481:       auto next_cost = cost + e.cost;
482:       if(dist[e.to] <= next_cost) continue;
483:       dist[e.to] = next_cost;
484:       que.emplace(dist[e.to], e.to);
```

```
  485:     }
  486:   }
  487:   return dist;
  488: }
  489:
  490:
  491: ####################################################
  492: ################## template.cpp ##################
  493: ####################################################
  494:
  495: template< typename T >
  496: struct edge {
  497:   int src, to;
  498:   T cost;
  499:
  500:   edge(int to, T cost) : src(-1), to(to), cost(cost) {}
  501:
  502:   edge(int src, int to, T cost) : src(src), to(to), cost(cost) {}
  503:
  504:   edge &operator=(const int &x) {
  505:     to = x;
  506:     return *this;
  507:   }
  508:
  509:   operator int() const { return to; }
  510: };
  511:
  512: template< typename T >
  513: using Edges = vector< edge< T > >;
  514: template< typename T >
  515: using WeightedGraph = vector< Edges< T > >;
  516: using UnWeightedGraph = vector< vector< int > >;
  517: template< typename T >
  518: using Matrix = vector< vector< T > >;
  519:
  520:
  521: ####################################################
  522: ############# bipartite-matching.cpp #############
  523: ####################################################
  524:
  525: struct BipartiteMatching {
  526:   vector< vector< int > > graph;
  527:   vector< int > match, alive, used;
  528:   int timestamp;
  529:
  530:   BipartiteMatching(int n) : graph(n), alive(n, 1), used(n, 0), match(n, -1), time
stamp(0) {}
  531:
  532:   void add_edge(int u, int v) {
  533:     graph[u].push_back(v);
  534:     graph[v].push_back(u);
  535:   }
  536:
  537:   bool dfs(int idx) {
  538:     used[idx] = timestamp;
  539:     for(auto &to : graph[idx]) {
  540:       int to_match = match[to];
  541:       if(alive[to] == 0) continue;
  542:       if(to_match == -1 || (used[to_match] != timestamp && dfs(to_match))) {
  543:         match[idx] = to;
  544:         match[to] = idx;
  545:         return true;
  546:       }
  547:     }
  548:     return false;
  549:   }
  550:
  551:   int bipartite_matching() {
  552:     int ret = 0;
  553:     for(int i = 0; i < graph.size(); i++) {
```

```
554:       if(alive[i] == 0) continue;
555:       if(match[i] == -1) {
556:         ++timestamp;
557:         ret += dfs(i);
558:       }
559:     }
560:     return ret;
561:   }
562:
563:   void output() {
564:     for(int i = 0; i < graph.size(); i++) {
565:       if(i < match[i]) {
566:         cout << i << "-" << match[i] << endl;
567:       }
568:     }
569:   }
570: };
571:
572:
573:
574: ######################################################
575: ############### warshall-floyd.cpp ###############
576: ######################################################
577:
578: template< typename T >
579: void warshall_floyd(Matrix< T > &g, T INF) {
580:   for(int k = 0; k < g.size(); k++) {
581:     for(int i = 0; i < g.size(); i++) {
582:       for(int j = 0; j < g.size(); j++) {
583:         if(g[i][k] == INF || g[k][j] == INF) continue;
584:         g[i][j] = min(g[i][j], g[i][k] + g[k][j]);
585:       }
586:     }
587:   }
588: }
589:
590:
591: ######################################################
592: ############### gabow-edmonds.cpp ################
593: ######################################################
594:
595: // https://qiita.com/Kutimoti_T/items/5b579773e0a24d650bdf
596: struct GabowEdmonds {
597:
598:   struct edge {
599:     int to, idx;
600:   };
601:
602:   vector< vector< edge > > g;
603:   vector< pair< int, int > > edges;
604:   vector< int > mate, label, first;
605:   queue< int > que;
606:
607:   GabowEdmonds(int n) : g(n + 1), mate(n + 1), label(n + 1, -1), first(n + 1) {}
608:
609:   void add_edge(int u, int v) {
610:     ++u, ++v;
611:     g[u].push_back((edge) {v, (int) (edges.size() + g.size())});
612:     g[v].push_back((edge) {u, (int) (edges.size() + g.size())});
613:     edges.emplace_back(u, v);
614:   }
615:
616:   int find(int x) {
617:     if(label[first[x]] < 0) return first[x];
618:     first[x] = find(first[x]);
619:     return first[x];
620:   }
621:
622:   void rematch(int v, int w) {
623:     int t = mate[v];
```

```
624:     mate[v] = w;
625:     if(mate[t] != v) return;
626:     if(label[v] < g.size()) {
627:       mate[t] = label[v];
628:       rematch(label[v], t);
629:     } else {
630:       int x = edges[label[v] - g.size()].first;
631:       int y = edges[label[v] - g.size()].second;
632:       rematch(x, y);
633:       rematch(y, x);
634:     }
635:   }
636:
637:   void assign_label(int x, int y, int num) {
638:     int r = find(x);
639:     int s = find(y);
640:     int join = 0;
641:     if(r == s) return;
642:     label[r] = -num;
643:     label[s] = -num;
644:     while(true) {
645:       if(s != 0) swap(r, s);
646:       r = find(label[mate[r]]);
647:       if(label[r] == -num) {
648:         join = r;
649:         break;
650:       }
651:       label[r] = -num;
652:     }
653:     int v = first[x];
654:     while(v != join) {
655:       que.push(v);
656:       label[v] = num;
657:       first[v] = join;
658:       v = first[label[mate[v]]];
659:     }
660:     v = first[y];
661:     while(v != join) {
662:       que.push(v);
663:       label[v] = num;
664:       first[v] = join;
665:       v = first[label[mate[v]]];
666:     }
667:   }
668:
669:   bool augment_check(int u) {
670:     que = queue< int >();
671:     first[u] = 0;
672:     label[u] = 0;
673:     que.push(u);
674:     while(!que.empty()) {
675:       int x = que.front();
676:       que.pop();
677:       for(auto e : g[x]) {
678:         int y = e.to;
679:         if(mate[y] == 0 && y != u) {
680:           mate[y] = x;
681:           rematch(x, y);
682:           return true;
683:         } else if(label[y] >= 0) {
684:           assign_label(x, y, e.idx);
685:         } else if(label[mate[y]] < 0) {
686:           label[mate[y]] = x;
687:           first[mate[y]] = y;
688:           que.push(mate[y]);
689:         }
690:       }
691:     }
692:     return false;
693:   }
```

```
694:
695:   vector< pair< int, int > > max_matching() {
696:     for(int i = 1; i < g.size(); i++) {
697:       if(mate[i] != 0) continue;
698:       if(augment_check(i)) label.assign(g.size(), -1);
699:     }
700:     vector< pair< int, int > > ret;
701:     for(int i = 1; i < g.size(); i++) {
702:       if(i < mate[i]) ret.emplace_back(i - 1, mate[i] - 1);
703:     }
704:     return ret;
705:   }
706: };
707:
708:
709: ##################################################
710: ############ maxflow-lower-bound.cpp #############
711: ##################################################
712:
713: template< typename flow_t, template< typename > class F >
714: struct MaxFlowLowerBound {
715:   F< flow_t > flow;
716:   vector< flow_t > in, up;
717:   typename F< flow_t >::edge *latte, *malta;
718:   int X, Y, V;
719:   flow_t sum;
720:
721:   MaxFlowLowerBound(int V) : V(V), flow(V + 2), X(V), Y(V + 1), sum(0), in(V) {}
722:
723:   void add_edge(int from, int to, flow_t low, flow_t high) {
724:     assert(from != to);
725:     flow.add_edge(from, to, high - low, up.size());
726:     in[from] -= low;
727:     in[to] += low;
728:     up.emplace_back(high);
729:   }
730:
731:   void build() {
732:     for(int i = 0; i < V; i++) {
733:       if(in[i] > 0) {
734:         flow.add_edge(X, i, in[i]);
735:         sum += in[i];
736:       } else if(in[i] < 0) {
737:         flow.add_edge(i, Y, -in[i]);
738:       }
739:     }
740:   }
741:
742:   bool can_flow(int s, int t) {
743:     assert(s != t);
744:     flow.add_edge(t, s, flow.INF);
745:     latte = &flow.graph[t].back();
746:     malta = &flow.graph[s].back();
747:     return can_flow();
748:   }
749:
750:   bool can_flow() {
751:     build();
752:     auto ret = flow.max_flow(X, Y);
753:     return ret >= sum;
754:   }
755:
756:   flow_t max_flow(int s, int t) {
757:     if(can_flow(s, t)) {
758:       return flow.max_flow(s, t);
759:     } else {
760:       return -1;
761:     }
762:   }
763:
```

```
  764:   flow_t min_flow(int s, int t) {
  765:     if(can_flow(s, t)) {
  766:       auto ret = flow.INF - latte->cap;
  767:       latte->cap = malta->cap = 0;
  768:       return ret - flow.max_flow(t, s);
  769:     } else {
  770:       return -1;
  771:     }
  772:   }
  773:
  774:   void output(int M) {
  775:     vector< flow_t > ans(M);
  776:     for(int i = 0; i < flow.graph.size(); i++) {
  777:       for(auto &e : flow.graph[i]) {
  778:         if(!e.isrev && ~e.idx) ans[e.idx] = up[e.idx] - e.cap;
  779:       }
  780:     }
  781:     for(auto &p : ans) cout << p << endl;
  782:   }
  783: };
  784:
  785:
  786: ######################################################
  787: ################ primal-dual.cpp #################
  788: ######################################################
  789:
  790: template< typename flow_t, typename cost_t >
  791: struct PrimalDual {
  792:   const cost_t INF;
  793:
  794:   struct edge {
  795:     int to;
  796:     flow_t cap;
  797:     cost_t cost;
  798:     int rev;
  799:     bool isrev;
  800:   };
  801:   vector< vector< edge > > graph;
  802:   vector< cost_t > potential, min_cost;
  803:   vector< int > prevv, preve;
  804:
  805:   PrimalDual(int V) : graph(V), INF(numeric_limits< cost_t >::max()) {}
  806:
  807:   void add_edge(int from, int to, flow_t cap, cost_t cost) {
  808:     graph[from].emplace_back((edge) {to, cap, cost, (int) graph[to].size(), false}
);
  809:     graph[to].emplace_back((edge) {from, 0, -cost, (int) graph[from].size() - 1, t
rue});
  810:   }
  811:
  812:   cost_t min_cost_flow(int s, int t, flow_t f) {
  813:     int V = (int) graph.size();
  814:     cost_t ret = 0;
  815:     using Pi = pair< cost_t, int >;
  816:     priority_queue< Pi, vector< Pi >, greater< Pi > > que;
  817:     potential.assign(V, 0);
  818:     preve.assign(V, -1);
  819:     prevv.assign(V, -1);
  820:
  821:     while(f > 0) {
  822:       min_cost.assign(V, INF);
  823:       que.emplace(0, s);
  824:       min_cost[s] = 0;
  825:       while(!que.empty()) {
  826:         Pi p = que.top();
  827:         que.pop();
  828:         if(min_cost[p.second] < p.first) continue;
  829:         for(int i = 0; i < graph[p.second].size(); i++) {
  830:           edge &e = graph[p.second][i];
  831:           cost_t nextCost = min_cost[p.second] + e.cost + potential[p.second] - po
```

```
tential[e.to];
  832:           if(e.cap > 0 && min_cost[e.to] > nextCost) {
  833:             min_cost[e.to] = nextCost;
  834:             prevv[e.to] = p.second, preve[e.to] = i;
  835:             que.emplace(min_cost[e.to], e.to);
  836:           }
  837:         }
  838:       }
  839:       if(min_cost[t] == INF) return -1;
  840:       for(int v = 0; v < V; v++) potential[v] += min_cost[v];
  841:       flow_t addflow = f;
  842:       for(int v = t; v != s; v = prevv[v]) {
  843:         addflow = min(addflow, graph[prevv[v]][preve[v]].cap);
  844:       }
  845:       f -= addflow;
  846:       ret += addflow * potential[t];
  847:       for(int v = t; v != s; v = prevv[v]) {
  848:         edge &e = graph[prevv[v]][preve[v]];
  849:         e.cap -= addflow;
  850:         graph[v][e.rev].cap += addflow;
  851:       }
  852:     }
  853:     return ret;
  854:   }
  855:
  856:   void output() {
  857:     for(int i = 0; i < graph.size(); i++) {
  858:       for(auto &e : graph[i]) {
  859:         if(e.isrev) continue;
  860:         auto &rev_e = graph[e.to][e.rev];
  861:         cout << i << "->" << e.to << " (flow: " << rev_e.cap << "/" << rev_e.cap +
 e.cap << ")" << endl;
  862:       }
  863:     }
  864:   }
  865: };
  866:
  867:
  868: ######################################################
  869: ################ bellman-ford.cpp ################
  870: ######################################################
  871:
  872: template< typename T >
  873: vector< T > bellman_ford(Edges< T > &edges, int V, int s) {
  874:   const auto INF = numeric_limits< T >::max();
  875:   vector< T > dist(V, INF);
  876:   dist[s] = 0;
  877:   for(int i = 0; i < V - 1; i++) {
  878:     for(auto &e : edges) {
  879:       if(dist[e.src] == INF) continue;
  880:       dist[e.to] = min(dist[e.to], dist[e.src] + e.cost);
  881:     }
  882:   }
  883:   for(auto &e : edges) {
  884:     if(dist[e.src] == INF) continue;
  885:     if(dist[e.src] + e.cost < dist[e.to]) return vector< T >();
  886:   }
  887:   return dist;
  888: }
  889:
  890:
  891:
  892: ######################################################
  893: ################ hungarian.cpp ################
  894: ######################################################
  895:
  896: template< typename T >
  897: T hungarian(Matrix< T > &A) {
  898:   const T infty = numeric_limits< T >::max();
  899:   const int N = (int) A.size();
```

```
  const int M = (int) A[0].size();
  vector< int > P(M), way(M);
  vector< T > U(N, 0), V(M, 0), minV;
  vector< bool > used;

  for(int i = 1; i < N; i++) {
    P[0] = i;
    minV.assign(M, infty);
    used.assign(M, false);
    int j0 = 0;
    while(P[j0] != 0) {
      int i0 = P[j0], j1 = 0;
      used[j0] = true;
      T delta = infty;
      for(int j = 1; j < M; j++) {
        if(used[j]) continue;
        T curr = A[i0][j] - U[i0] - V[j];
        if(curr < minV[j]) minV[j] = curr, way[j] = j0;
        if(minV[j] < delta) delta = minV[j], j1 = j;
      }
      for(int j = 0; j < M; j++) {
        if(used[j]) U[P[j]] += delta, V[j] -= delta;
        else minV[j] -= delta;
      }
      j0 = j1;
    }
    do {
      P[j0] = P[way[j0]];
      j0 = way[j0];
    } while(j0 != 0);
  }
  return -V[0];
}


////////////////////////////////////////////////////
////////// maximum-independent-set.cpp ////////////
////////////////////////////////////////////////////

template< typename T >
vector< int > maximum_independent_set(const Matrix< T > &g, int trial = 1000000) {

  int N = (int) g.size();
  vector< uint64_t > bit(N);

  assert(N <= 64);
  for(int i = 0; i < N; i++) {
    for(int j = 0; j < N; j++) {
      if(i != j) {
        assert(g[i][j] == g[j][i]);
        if(g[i][j]) bit[i] |= uint64_t(1) << j;
      }
    }
  }

  vector< int > ord(N);
  iota(begin(ord), end(ord), 0);
  mt19937 mt(chrono::steady_clock::now().time_since_epoch().count());
  int ret = 0;
  uint64_t ver;
  for(int i = 0; i < trial; i++) {
    shuffle(begin(ord), end(ord), mt);
    uint64_t used = 0;
    int add = 0;
    for(int j : ord) {
      if(used & bit[j]) continue;
      used |= uint64_t(1) << j;
      ++add;
    }
    if(ret < add) {
```

```
  970:       ret = add;
  971:       ver = used;
  972:     }
  973:   }
  974:   vector< int > ans;
  975:   for(int i = 0; i < N; i++) {
  976:     if((ver >> i) & 1) ans.emplace_back(i);
  977:   }
  978:   return ans;
  979: }
  980:
  981:
  982: ##################################################
  983: ################ push-relabel.cpp ################
  984: ##################################################
  985:
  986: template< typename flow_t >
  987: struct PushRelabel {
  988:   const flow_t INF;
  989:
  990:   struct edge {
  991:     int to;
  992:     flow_t cap;
  993:     int rev;
  994:     bool isrev;
  995:     int idx;
  996:   };
  997:   vector< vector< edge > > graph;
  998:   vector< flow_t > ex;
  999:   int relabels, high;
 1000:   vector< int > cnt, h;
 1001:   vector< vector< int > > hs;
 1002:
 1003:   PushRelabel(int V) : graph(V), INF(numeric_limits< flow_t >::max()), hs(V + 1),
high(0) {}
 1004:
 1005:
 1006:   void add_edge(int from, int to, flow_t cap, int idx = -1) {
 1007:     graph[from].emplace_back((edge) {to, cap, (int) graph[to].size(), false, idx})
;
 1008:     graph[to].emplace_back((edge) {from, 0, (int) graph[from].size() - 1, true, id
x});
 1009:   }
 1010:
 1011:   void update_height(int idx, int nxt_height) {
 1012:     ++relabels;
 1013:     if(h[idx] != graph.size() + 1) {
 1014:       --cnt[h[idx]];
 1015:     }
 1016:     h[idx] = nxt_height;
 1017:     if(h[idx] != graph.size() + 1) {
 1018:       high = nxt_height;
 1019:       ++cnt[nxt_height];
 1020:       if(ex[idx] > 0) hs[nxt_height].emplace_back(idx);
 1021:     }
 1022:   }
 1023:
 1024:   void global_relabel(int idx) {
 1025:     for(int i = 0; i <= high; i++) hs[i].clear();
 1026:     relabels = 0;
 1027:     high = 0;
 1028:     h.assign(graph.size(), graph.size() + 1);
 1029:     cnt.assign(graph.size(), 0);
 1030:     queue< int > que;
 1031:     que.emplace(idx);
 1032:     h[idx] = 0;
 1033:     while(que.size()) {
 1034:       int p = que.front();
 1035:       que.pop();
 1036:       for(auto &e : graph[p]) {
```

```
1037:         if(h[e.to] == graph.size() + 1 && graph[e.to][e.rev].cap > 0) {
1038:           que.emplace(e.to);
1039:           high = h[p] + 1;
1040:           update_height(e.to, high);
1041:         }
1042:       }
1043:     }
1044:   }
1045:
1046:
1047:   void push(int idx, edge &e) {
1048:     if(h[e.to] == graph.size() + 1) return;
1049:     if(ex[e.to] == 0) {
1050:       hs[h[e.to]].emplace_back(e.to);
1051:     }
1052:     flow_t df = min(ex[idx], e.cap);
1053:     e.cap -= df;
1054:     graph[e.to][e.rev].cap += df;
1055:     ex[idx] -= df;
1056:     ex[e.to] += df;
1057:   }
1058:
1059:   void discharge(int idx) {
1060:     int next_height = (int) graph.size() + 1;
1061:     for(auto &&e : graph[idx]) {
1062:       if(e.cap > 0) {
1063:         if(h[idx] == h[e.to] + 1) {
1064:           push(idx, e);
1065:           if(ex[idx] <= 0) return;
1066:         } else {
1067:           next_height = min(next_height, h[e.to] + 1);
1068:         }
1069:       }
1070:     }
1071:     if(cnt[h[idx]] > 1) {
1072:       update_height(idx, next_height);
1073:     } else {
1074:       for(; high >= h[idx]; hs[high--].clear()) {
1075:         for(int j : hs[high]) update_height(j, graph.size() + 1);
1076:       }
1077:     }
1078:   }
1079:
1080:   flow_t max_flow(int s, int t) {
1081:     ex.assign(graph.size(), 0);
1082:     ex[s] = INF;
1083:     ex[t] = -INF;
1084:     global_relabel(t);
1085:     for(auto &e : graph[s]) push(s, e);
1086:     for(; high >= 0; high--) {
1087:       while(!hs[high].empty()) {
1088:         int idx = hs[high].back();
1089:         hs[high].pop_back();
1090:         discharge(idx);
1091:         if(relabels >= graph.size() * 4) global_relabel(t);
1092:       }
1093:     }
1094:     return ex[t] + INF;
1095:   }
1096:
1097:   void output() {
1098:     for(int i = 0; i < graph.size(); i++) {
1099:       for(auto &e : graph[i]) {
1100:         if(e.isrev) continue;
1101:         auto &rev_e = graph[e.to][e.rev];
1102:         cout << i << "->" << e.to << " (flow: " << rev_e.cap << "/" << e.cap + rev
_e.cap << ")" << endl;
1103:       }
1104:     }
1105:   }
```

```
 1106: };
 1107:
 1108:
 1109: ////////////////////////////////////////////////
 1110: ////////////////// kruskal.cpp //////////////////
 1111: ////////////////////////////////////////////////
 1112:
 1113: template< typename T >
 1114: T kruskal(Edges< T > &edges, int V) {
 1115:   sort(begin(edges), end(edges), [](const edge< T > &a, const edge< T > &b) {
 1116:     return (a.cost < b.cost);
 1117:   });
 1118:   UnionFind tree(V);
 1119:   T ret = 0;
 1120:   for(auto &e : edges) {
 1121:     if(tree.unite(e.src, e.to)) ret += e.cost;
 1122:   }
 1123:   return (ret);
 1124: }
 1125:
 1126:
 1127: ////////////////////////////////////////////////
 1128: ////////////// ford-fulkerson.cpp //////////////
 1129: ////////////////////////////////////////////////
 1130:
 1131: template< typename flow_t >
 1132: struct FordFulkerson {
 1133:   struct edge {
 1134:     int to;
 1135:     flow_t cap;
 1136:     int rev;
 1137:     bool isrev;
 1138:     int idx;
 1139:   };
 1140:
 1141:   vector< vector< edge > > graph;
 1142:   vector< int > used;
 1143:   const flow_t INF;
 1144:   int timestamp;
 1145:
 1146:   FordFulkerson(int n) : INF(numeric_limits< flow_t >::max()), timestamp(0) {
 1147:     graph.resize(n);
 1148:     used.assign(n, -1);
 1149:   }
 1150:
 1151:   void add_edge(int from, int to, flow_t cap, int idx = -1) {
 1152:     graph[from].emplace_back((edge) {to, cap, (int) graph[to].size(), false, idx})
;
 1153:     graph[to].emplace_back((edge) {from, 0, (int) graph[from].size() - 1, true, id
x});
 1154:   }
 1155:
 1156:   flow_t dfs(int idx, const int t, flow_t flow) {
 1157:     if(idx == t) return flow;
 1158:     used[idx] = timestamp;
 1159:     for(auto &e : graph[idx]) {
 1160:       if(e.cap > 0 && used[e.to] != timestamp) {
 1161:         flow_t d = dfs(e.to, t, min(flow, e.cap));
 1162:         if(d > 0) {
 1163:           e.cap -= d;
 1164:           graph[e.to][e.rev].cap += d;
 1165:           return d;
 1166:         }
 1167:       }
 1168:     }
 1169:     return 0;
 1170:   }
 1171:
 1172:   flow_t max_flow(int s, int t) {
 1173:     flow_t flow = 0;
```

```
 1174:     for(flow_t f; (f = dfs(s, t, INF)) > 0; timestamp++) {
 1175:       flow += f;
 1176:     }
 1177:     return flow;
 1178:   }
 1179:
 1180:   void output() {
 1181:     for(int i = 0; i < graph.size(); i++) {
 1182:       for(auto &e : graph[i]) {
 1183:         if(e.isrev) continue;
 1184:         auto &rev_e = graph[e.to][e.rev];
 1185:         cout << i << "->" << e.to << " (flow: " << rev_e.cap << "/" << e.cap + rev
_e.cap << ")" << endl;
 1186:       }
 1187:     }
 1188:   }
 1189: };
 1190:
 1191:
 1192: ######################################################
 1193: ########## offline-dag-reachability.cpp ##########
 1194: ######################################################
 1195:
 1196: template< typename G >
 1197: vector< int > offline_dag_reachability(const G &g, vector< pair< int, int > > &qs)
 {
 1198:   const int N = (int) g.size();
 1199:   const int Q = (int) qs.size();
 1200:   auto ord = topological_sort(g);
 1201:   vector< int > ans(Q);
 1202:   for(int l = 0; l < Q; l += 64) {
 1203:     int r = min(Q, l + 64);
 1204:     vector< int64_t > dp(N);
 1205:     for(int k = l; k < r; k++) {
 1206:       dp[qs[k].first] |= int64_t(1) << (k - l);
 1207:     }
 1208:     for(auto &idx : ord) {
 1209:       for(auto &to : g[idx]) dp[to] |= dp[idx];
 1210:     }
 1211:     for(int k = l; k < r; k++) {
 1212:       ans[k] = (dp[qs[k].second] >> (k - l)) & 1;
 1213:     }
 1214:   }
 1215:   return ans;
 1216: }
 1217:
 1218:
 1219: ######################################################
 1220: ############## chromatic-number.cpp ##############
 1221: ######################################################
 1222:
 1223: int chromatic_number(const Matrix< bool > &g) {
 1224:   int N = (int) g.size();
 1225:   vector< int > es(N);
 1226:   for(int i = 0; i < g.size(); i++) {
 1227:     for(int j = 0; j < g.size(); j++) {
 1228:       es[i] |= g[i][j] << j;
 1229:     }
 1230:   }
 1231:   int ret = N;
 1232:   for(int d : {7, 11, 21}) {
 1233:     int mod = 1e9 + d;
 1234:     vector< int > ind(1 << N), aux(1 << N, 1);
 1235:     ind[0] = 1;
 1236:     for(int S = 1; S < 1 << N; S++) {
 1237:       int u = __builtin_ctz(S);
 1238:       ind[S] = ind[S ^ (1 << u)] + ind[(S ^ (1 << u)) & ~es[u]];
 1239:     }
 1240:     for(int i = 1; i < ret; i++) {
 1241:       int64_t all = 0;
```

```
1242:       for(int j = 0; j < 1 << N; j++) {
1243:         int S = j ^(j >> 1);
1244:         aux[S] = (1LL * aux[S] * ind[S]) % mod;
1245:         all += j & 1 ? aux[S] : mod - aux[S];
1246:       }
1247:       if(all % mod) ret = i;
1248:     }
1249:   }
1250:   return ret;
1251: }
1252:
1253:
1254:
1255: ######################################################
1256: ################## grid-bfs.cpp ######################
1257: ######################################################
1258:
1259: vector< vector< int > > grid_bfs(vector< string > &s, char start, const string &wa
ll = "#") {
1260:   const int vx[] = {0, 1, 0, -1}, vy[] = {1, 0, -1, 0};
1261:   vector< vector< int > > min_cost(s.size(), vector< int >(s[0].size(), -1));
1262:   queue< pair< int, int > > que;
1263:   for(int i = 0; i < s.size(); i++) {
1264:     for(int j = 0; j < s[i].size(); j++) {
1265:       if(s[i][j] == start) {
1266:         que.emplace(i, j);
1267:         min_cost[i][j] = 0;
1268:       }
1269:     }
1270:   }
1271:   while(!que.empty()) {
1272:     auto p = que.front();
1273:     que.pop();
1274:     for(int i = 0; i < 4; i++) {
1275:       int ny = p.first + vy[i], nx = p.second + vx[i];
1276:       if(nx < 0 || ny < 0 || nx >= s[0].size() || ny >= s.size()) continue;
1277:       if(min_cost[ny][nx] != -1) continue;
1278:       if(wall.find(s[ny][nx]) != string::npos) continue;
1279:       min_cost[ny][nx] = min_cost[p.first][p.second] + 1;
1280:       que.emplace(ny, nx);
1281:     }
1282:   }
1283:   return min_cost;
1284: }
1285:
1286:
1287: ######################################################
1288: ####### two-edge-connected-components.cpp ########
1289: ######################################################
1290:
1291: template< typename G >
1292: struct TwoEdgeConnectedComponents : LowLink< G > {
1293:   using LL = LowLink< G >;
1294:   vector< int > comp;
1295:
1296:   TwoEdgeConnectedComponents(const G &g) : LL(g) {}
1297:
1298:   int operator[](const int &k) {
1299:     return comp[k];
1300:   }
1301:
1302:   void dfs(int idx, int par, int &k) {
1303:     if(~par && this->ord[par] >= this->low[idx]) comp[idx] = comp[par];
1304:     else comp[idx] = k++;
1305:     for(auto &to : this->g[idx]) {
1306:       if(comp[to] == -1) dfs(to, idx, k);
1307:     }
1308:   }
1309:
1310:   void build(UnWeightedGraph &t) {
```

```
 1311:     LL::build();
 1312:     comp.assign(this->g.size(), -1);
 1313:     int k = 0;
 1314:     for(int i = 0; i < comp.size(); i++) {
 1315:       if(comp[i] == -1) dfs(i, -1, k);
 1316:     }
 1317:     t.resize(k);
 1318:     for(auto &e : this->bridge) {
 1319:       int x = comp[e.first], y = comp[e.second];
 1320:       t[x].push_back(y);
 1321:       t[y].push_back(x);
 1322:     }
 1323:   }
 1324: };
 1325:
 1326:
 1327: ##################################################
 1328: ########### dinic-capacity-scaling.cpp ###########
 1329: ##################################################
 1330:
 1331: template< typename flow_t >
 1332: struct DinicCapacityScaling {
 1333:
 1334:   const flow_t INF;
 1335:
 1336:   struct edge {
 1337:     int to;
 1338:     flow_t cap;
 1339:     int rev;
 1340:     bool isrev;
 1341:   };
 1342:
 1343:   vector< vector< edge > > graph;
 1344:   vector< int > min_cost, iter;
 1345:   flow_t max_cap;
 1346:
 1347:   DinicCapacityScaling(int V) : INF(numeric_limits< flow_t >::max()), graph(V), ma
x_cap(0) {}
 1348:
 1349:   void add_edge(int from, int to, flow_t cap) {
 1350:     max_cap = max(max_cap, cap);
 1351:     graph[from].emplace_back((edge) {to, cap, (int) graph[to].size(), false});
 1352:     graph[to].emplace_back((edge) {from, 0, (int) graph[from].size() - 1, true});
 1353:   }
 1354:
 1355:   bool bfs(int s, int t, const flow_t &base) {
 1356:     min_cost.assign(graph.size(), -1);
 1357:     queue< int > que;
 1358:     min_cost[s] = 0;
 1359:     que.push(s);
 1360:     while(!que.empty() && min_cost[t] == -1) {
 1361:       int p = que.front();
 1362:       que.pop();
 1363:       for(auto &e : graph[p]) {
 1364:         if(e.cap >= base && min_cost[e.to] == -1) {
 1365:           min_cost[e.to] = min_cost[p] + 1;
 1366:           que.push(e.to);
 1367:         }
 1368:       }
 1369:     }
 1370:     return min_cost[t] != -1;
 1371:   }
 1372:
 1373:   flow_t dfs(int idx, const int t, const flow_t base, flow_t flow) {
 1374:     if(idx == t) return flow;
 1375:     flow_t sum = 0;
 1376:     for(int &i = iter[idx]; i < graph[idx].size(); i++) {
 1377:       edge &e = graph[idx][i];
 1378:       if(e.cap >= base && min_cost[idx] < min_cost[e.to]) {
 1379:         flow_t d = dfs(e.to, t, base, min(flow - sum, e.cap));
```

```
1380:         if(d > 0) {
1381:           e.cap -= d;
1382:           graph[e.to][e.rev].cap += d;
1383:           sum += d;
1384:           if(flow - sum < base) break;
1385:         }
1386:       }
1387:     }
1388:     return sum;
1389:   }
1390:
1391:   flow_t max_flow(int s, int t) {
1392:     if(max_cap == flow_t(0)) return flow_t(0);
1393:     flow_t flow = 0;
1394:     for(int i = 63 - __builtin_clzll(max_cap); i >= 0; i--) {
1395:       flow_t now = flow_t(1) << i;
1396:       while(bfs(s, t, now)) {
1397:         iter.assign(graph.size(), 0);
1398:         flow += dfs(s, t, now, INF);
1399:       }
1400:     }
1401:     return flow;
1402:   }
1403:
1404:   void output() {
1405:     for(int i = 0; i < graph.size(); i++) {
1406:       for(auto &e : graph[i]) {
1407:         if(e.isrev) continue;
1408:         auto &rev_e = graph[e.to][e.rev];
1409:         cout << i << "->" << e.to << " (flow: " << rev_e.cap << "/" << e.cap + rev
_e.cap << ")" << endl;
1410:       }
1411:     }
1412:   }
1413: };
1414:
1415:
1416:
1417: ######################################################
1418: ########## bi-connected-components.cpp ###########
1419: ######################################################
1420:
1421: template< typename G >
1422: struct BiConnectedComponents : LowLink< G > {
1423:   using LL = LowLink< G >;
1424:
1425:   vector< int > used;
1426:   vector< vector< pair< int, int > > > bc;
1427:   vector< pair< int, int > > tmp;
1428:
1429:   BiConnectedComponents(const G &g) : LL(g) {}
1430:
1431:   void dfs(int idx, int par) {
1432:     used[idx] = true;
1433:     for(auto &to : this->g[idx]) {
1434:       if(to == par) continue;
1435:       if(!used[to] || this->ord[to] < this->ord[idx]) {
1436:         tmp.emplace_back(minmax(idx, to));
1437:       }
1438:       if(!used[to]) {
1439:         dfs(to, idx);
1440:         if(this->low[to] >= this->ord[idx]) {
1441:           bc.emplace_back();
1442:           for(;;) {
1443:             auto e = tmp.back();
1444:             bc.back().emplace_back(e);
1445:             tmp.pop_back();
1446:             if(e.first == min(idx, to) && e.second == max(idx, to)) {
1447:               break;
1448:             }
```

```
 1449:           }
 1450:         }
 1451:       }
 1452:     }
 1453:   }
 1454:
 1455:   void build() override {
 1456:     LL::build();
 1457:     used.assign(this->g.size(), 0);
 1458:     for(int i = 0; i < used.size(); i++) {
 1459:       if(!used[i]) dfs(i, -1);
 1460:     }
 1461:   }
 1462: };
 1463:
 1464:
 1465: ##################################################
 1466: ####### strongly-connected-components.cpp ########
 1467: ##################################################
 1468:
 1469: template< typename G >
 1470: struct StronglyConnectedComponents {
 1471:   const G &g;
 1472:   UnWeightedGraph gg, rg;
 1473:   vector< int > comp, order, used;
 1474:
 1475:   StronglyConnectedComponents(G &g) : g(g), gg(g.size()), rg(g.size()), comp(g.siz
e(), -1), used(g.size()) {
 1476:     for(int i = 0; i < g.size(); i++) {
 1477:       for(auto e : g[i]) {
 1478:         gg[i].emplace_back((int) e);
 1479:         rg[(int) e].emplace_back(i);
 1480:       }
 1481:     }
 1482:   }
 1483:
 1484:   int operator[](int k) {
 1485:     return comp[k];
 1486:   }
 1487:
 1488:   void dfs(int idx) {
 1489:     if(used[idx]) return;
 1490:     used[idx] = true;
 1491:     for(int to : gg[idx]) dfs(to);
 1492:     order.push_back(idx);
 1493:   }
 1494:
 1495:   void rdfs(int idx, int cnt) {
 1496:     if(comp[idx] != -1) return;
 1497:     comp[idx] = cnt;
 1498:     for(int to : rg[idx]) rdfs(to, cnt);
 1499:   }
 1500:
 1501:   void build(UnWeightedGraph &t) {
 1502:     for(int i = 0; i < gg.size(); i++) dfs(i);
 1503:     reverse(begin(order), end(order));
 1504:     int ptr = 0;
 1505:     for(int i : order) if(comp[i] == -1) rdfs(i, ptr), ptr++;
 1506:
 1507:     t.resize(ptr);
 1508:     for(int i = 0; i < g.size(); i++) {
 1509:       for(auto &to : g[i]) {
 1510:         int x = comp[i], y = comp[to];
 1511:         if(x == y) continue;
 1512:         t[x].push_back(y);
 1513:       }
 1514:     }
 1515:   }
 1516: };
 1517:
```

```
 1518:
 1519: ####################################################
 1520: ############### dominator-tree.cpp ################
 1521: ####################################################
 1522:
 1523: template< typename G >
 1524: struct DominatorTree {
 1525:
 1526:   struct UnionFind {
 1527:     const vector< int > &semi;
 1528:     vector< int > par, m;
 1529:
 1530:     UnionFind(const vector< int > &semi) : semi(semi), par(semi.size()), m(semi.si
ze()) {
 1531:       iota(begin(par), end(par), 0);
 1532:       iota(begin(m), end(m), 0);
 1533:     }
 1534:
 1535:     int find(int v) {
 1536:       if(par[v] == v) return v;
 1537:       int r = find(par[v]);
 1538:       if(semi[m[v]] > semi[m[par[v]]]) m[v] = m[par[v]];
 1539:       return par[v] = r;
 1540:     }
 1541:
 1542:     int eval(int v) {
 1543:       find(v);
 1544:       return m[v];
 1545:     }
 1546:
 1547:     void link(int p, int c) {
 1548:       par[c] = p;
 1549:     }
 1550:   };
 1551:
 1552:   const G &g;
 1553:   vector< vector< int > > rg;
 1554:   vector< int > ord, par;
 1555:   vector< int > idom, semi;
 1556:   UnionFind uf;
 1557:
 1558:   DominatorTree(G &g) : g(g), rg(g.size()), par(g.size()), idom(g.size(), -1), sem
i(g.size(), -1), uf(semi) {
 1559:     ord.reserve(g.size());
 1560:   }
 1561:
 1562:
 1563:   void dfs(int idx) {
 1564:     semi[idx] = (int) ord.size();
 1565:     ord.emplace_back(idx);
 1566:     for(auto &to : g[idx]) {
 1567:       if(~semi[to]) continue;
 1568:       dfs(to);
 1569:       par[to] = idx;
 1570:     }
 1571:   }
 1572:
 1573:   void build(int root) {
 1574:     const int N = (int) g.size();
 1575:     dfs(root);
 1576:     for(int i = 0; i < N; i++) {
 1577:       for(auto &to : g[i]) {
 1578:         if(~semi[i]) rg[to].emplace_back(i);
 1579:       }
 1580:     }
 1581:
 1582:     vector< vector< int > > bucket(N);
 1583:     vector< int > U(N);
 1584:     for(int i = (int) ord.size() - 1; i >= 0; i--) {
 1585:       int x = ord[i];
```

```
1586:       for(int v : rg[x]) {
1587:         v = uf.eval(v);
1588:         if(semi[x] > semi[v]) semi[x] = semi[v];
1589:       }
1590:       bucket[ord[semi[x]]].emplace_back(x);
1591:       for(int v : bucket[par[x]]) U[v] = uf.eval(v);
1592:       bucket[par[x]].clear();
1593:       uf.link(par[x], x);
1594:     }
1595:     for(int i = 1; i < ord.size(); i++) {
1596:       int x = ord[i], u = U[x];
1597:       idom[x] = semi[x] == semi[u] ? semi[x] : idom[u];
1598:     }
1599:     for(int i = 1; i < ord.size(); i++) {
1600:       int x = ord[i];
1601:       idom[x] = ord[idom[x]];
1602:     }
1603:     idom[root] = root;
1604:   }
1605:
1606:   int operator[](const int &k) {
1607:     return idom[k];
1608:   }
1609: };
1610:
1611:
1612: ######################################################
1613: ################### prim.cpp #########################
1614: ######################################################
1615:
1616: template< typename T >
1617: T prim(WeightedGraph< T > &g) {
1618:   using Pi = pair< T, int >;
1619:
1620:   T total = 0;
1621:   vector< bool > used(g.size(), false);
1622:   priority_queue< Pi, vector< Pi >, greater< Pi > > que;
1623:   que.emplace(0, 0);
1624:   while(!que.empty()) {
1625:     auto p = que.top();
1626:     que.pop();
1627:     if(used[p.second]) continue;
1628:     used[p.second] = true;
1629:     total += p.first;
1630:     for(auto &e : g[p.second]) {
1631:       que.emplace(e.cost, e.to);
1632:     }
1633:   }
1634:   return total;
1635: }
1636:
1637:
1638: ######################################################
1639: ############### eulerian-trail.cpp ###################
1640: ######################################################
1641:
1642: template< typename T >
1643: vector< edge< T > > eulerian_path(Edges< T > es, int s, bool directed) {
1644:   int V = 0;
1645:   for(auto &e : es) V = max(V, max(e.to, e.src) + 1);
1646:   vector< vector< pair< edge< T >, int > > > g(V);
1647:   for(auto &e : es) {
1648:     int sz_to = (int) g[e.to].size();
1649:     g[e.src].emplace_back(e, sz_to);
1650:     if(!directed) {
1651:       int sz_src = (int) g[e.src].size() - 1;
1652:       swap(e.src, e.to);
1653:       g[e.src].emplace_back(e, sz_src);
1654:     }
1655:   }
```

```
 1656:   vector< edge< T > > ord;
 1657:   stack< pair< int, edge< T > > > st;
 1658:   st.emplace(s, edge< T >(-1, -1));
 1659:   while(st.size()) {
 1660:     int idx = st.top().first;
 1661:     if(g[idx].empty()) {
 1662:       ord.emplace_back(st.top().second);
 1663:       st.pop();
 1664:     } else {
 1665:       auto e = g[idx].back();
 1666:       g[idx].pop_back();
 1667:       if(e.second == -1) continue;
 1668:       if(!directed) g[e.first.to][e.second].second = -1;
 1669:       st.emplace(e.first.to, e.first);
 1670:     }
 1671:   }
 1672:   ord.pop_back();
 1673:   reverse(begin(ord), end(ord));
 1674:   if(ord.size() != es.size()) return {};
 1675:   return ord;
 1676: }
 1677:
 1678:
 1679: ##################################################
 1680: ################### dinic.cpp ####################
 1681: ##################################################
 1682:
 1683: template< typename flow_t >
 1684: struct Dinic {
 1685:   const flow_t INF;
 1686:
 1687:   struct edge {
 1688:     int to;
 1689:     flow_t cap;
 1690:     int rev;
 1691:     bool isrev;
 1692:     int idx;
 1693:   };
 1694:
 1695:   vector< vector< edge > > graph;
 1696:   vector< int > min_cost, iter;
 1697:
 1698:   Dinic(int V) : INF(numeric_limits< flow_t >::max()), graph(V) {}
 1699:
 1700:   void add_edge(int from, int to, flow_t cap, int idx = -1) {
 1701:     graph[from].emplace_back((edge) {to, cap, (int) graph[to].size(), false, idx})
;
 1702:     graph[to].emplace_back((edge) {from, 0, (int) graph[from].size() - 1, true, id
x});
 1703:   }
 1704:
 1705:   bool bfs(int s, int t) {
 1706:     min_cost.assign(graph.size(), -1);
 1707:     queue< int > que;
 1708:     min_cost[s] = 0;
 1709:     que.push(s);
 1710:     while(!que.empty() && min_cost[t] == -1) {
 1711:       int p = que.front();
 1712:       que.pop();
 1713:       for(auto &e : graph[p]) {
 1714:         if(e.cap > 0 && min_cost[e.to] == -1) {
 1715:           min_cost[e.to] = min_cost[p] + 1;
 1716:           que.push(e.to);
 1717:         }
 1718:       }
 1719:     }
 1720:     return min_cost[t] != -1;
 1721:   }
 1722:
 1723:   flow_t dfs(int idx, const int t, flow_t flow) {
```

```
 1724:     if(idx == t) return flow;
 1725:     for(int &i = iter[idx]; i < graph[idx].size(); i++) {
 1726:       edge &e = graph[idx][i];
 1727:       if(e.cap > 0 && min_cost[idx] < min_cost[e.to]) {
 1728:         flow_t d = dfs(e.to, t, min(flow, e.cap));
 1729:         if(d > 0) {
 1730:           e.cap -= d;
 1731:           graph[e.to][e.rev].cap += d;
 1732:           return d;
 1733:         }
 1734:       }
 1735:     }
 1736:     return 0;
 1737:   }
 1738:
 1739:   flow_t max_flow(int s, int t) {
 1740:     flow_t flow = 0;
 1741:     while(bfs(s, t)) {
 1742:       iter.assign(graph.size(), 0);
 1743:       flow_t f = 0;
 1744:       while((f = dfs(s, t, INF)) > 0) flow += f;
 1745:     }
 1746:     return flow;
 1747:   }
 1748:
 1749:   void output() {
 1750:     for(int i = 0; i < graph.size(); i++) {
 1751:       for(auto &e : graph[i]) {
 1752:         if(e.isrev) continue;
 1753:         auto &rev_e = graph[e.to][e.rev];
 1754:         cout << i << "->" << e.to << " (flow: " << rev_e.cap << "/" << e.cap + rev
_e.cap << ")" << endl;
 1755:       }
 1756:     }
 1757:   }
 1758: };
 1759:
 1760:
 1761: ////////////////////////////////////////////////
 1762: ////////////// topological-sort.cpp //////////////
 1763: ////////////////////////////////////////////////
 1764:
 1765: template< typename G >
 1766: vector< int > topological_sort(const G &g) {
 1767:   const int N = (int) g.size();
 1768:   vector< int > deg(N);
 1769:   for(int i = 0; i < N; i++) {
 1770:     for(auto &to : g[i]) ++deg[to];
 1771:   }
 1772:   stack< int > st;
 1773:   for(int i = 0; i < N; i++) {
 1774:     if(deg[i] == 0) st.emplace(i);
 1775:   }
 1776:   vector< int > ord;
 1777:   while(!st.empty()) {
 1778:     auto p = st.top();
 1779:     st.pop();
 1780:     ord.emplace_back(p);
 1781:     for(auto &to : g[p]) {
 1782:       if(--deg[to] == 0) st.emplace(to);
 1783:     }
 1784:   }
 1785:   return ord;
 1786: }
 1787:
 1788:
 1789: ////////////////////////////////////////////////
 1790: /////////////// chu-liu-edmond.cpp //////////////
 1791: ////////////////////////////////////////////////
 1792:
```

```
1793: template< typename T >
1794: struct MinimumSpanningTreeArborescence
1795: {
1796:   using Pi = pair< T, int >;
1797:   using Heap = SkewHeap< Pi, int >;
1798:   using Node = typename Heap::Node;
1799:   const Edges< T > &es;
1800:   const int V;
1801:   T INF;
1802:
1803:   MinimumSpanningTreeArborescence(const Edges< T > &es, int V) :
1804:       INF(numeric_limits< T >::max()), es(es), V(V) {}
1805:
1806:   T build(int start)
1807:   {
1808:     auto g = [](const Pi &a, const T &b) { return Pi(a.first + b, a.second); };
1809:     auto h = [](const T &a, const T &b) { return a + b; };
1810:     Heap heap(g, h);
1811:     vector< Node * > heaps(V, heap.makeheap());
1812:     for(auto &e : es) heap.push(heaps[e.to], {e.cost, e.src});
1813:     UnionFind uf(V);
1814:     vector< int > used(V, -1);
1815:     used[start] = start;
1816:
1817:     T ret = 0;
1818:     for(int s = 0; s < V; s++) {
1819:       stack< int > path;
1820:       for(int u = s; used[u] < 0;) {
1821:         path.push(u);
1822:         used[u] = s;
1823:         if(heap.empty(heaps[u])) return -1;
1824:         auto p = heap.top(heaps[u]);
1825:         ret += p.first;
1826:         heap.add(heaps[u], -p.first);
1827:         heap.pop(heaps[u]);
1828:         int v = uf.find(p.second);
1829:         if(used[v] == s) {
1830:           int w;
1831:           Node *nextheap = heap.makeheap();
1832:           do {
1833:             w = path.top();
1834:             path.pop();
1835:             nextheap = heap.merge(nextheap, heaps[w]);
1836:           } while(uf.unite(v, w));
1837:           heaps[uf.find(v)] = nextheap;
1838:           used[uf.find(v)] = -1;
1839:         }
1840:         u = uf.find(v);
1841:       }
1842:     }
1843:     return ret;
1844:   }
1845: };
1846:
1847:
1848: ##################################################
1849: ####### shortest-path-faster-algorithm.cpp #######
1850: ##################################################
1851:
1852: template< typename T >
1853: vector< T > shortest_path_faster_algorithm(WeightedGraph< T > &g, int s) {
1854:   const auto INF = numeric_limits< T >::max();
1855:   vector< T > dist(g.size(), INF);
1856:   vector< int > pending(g.size(), 0), times(g.size(), 0);
1857:   queue< int > que;
1858:
1859:   que.emplace(s);
1860:   pending[s] = true;
1861:   ++times[s];
1862:   dist[s] = 0;
```

```
1863:
1864:   while(!que.empty()) {
1865:     int p = que.front();
1866:     que.pop();
1867:     pending[p] = false;
1868:     for(auto &e : g[p]) {
1869:       T next_cost = dist[p] + e.cost;
1870:       if(next_cost >= dist[e.to]) continue;
1871:       dist[e.to] = next_cost;
1872:       if(!pending[e.to]) {
1873:         if(++times[e.to] >= g.size()) return vector< T >();
1874:         pending[e.to] = true;
1875:         que.emplace(e.to);
1876:       }
1877:     }
1878:   }
1879:   return dist;
1880: }
1881:
1882:
1883: ######################################################
1884: ############### hopcroft-karp.cpp ################
1885: ######################################################
1886:
1887: struct HopcroftKarp {
1888:   vector< vector< int > > graph;
1889:   vector< int > dist, match;
1890:   vector< bool > used, vv;
1891:
1892:   HopcroftKarp(int n, int m) : graph(n), match(m, -1), used(n) {}
1893:
1894:   void add_edge(int u, int v) {
1895:     graph[u].push_back(v);
1896:   }
1897:
1898:   void bfs() {
1899:     dist.assign(graph.size(), -1);
1900:     queue< int > que;
1901:     for(int i = 0; i < graph.size(); i++) {
1902:       if(!used[i]) {
1903:         que.emplace(i);
1904:         dist[i] = 0;
1905:       }
1906:     }
1907:
1908:     while(!que.empty()) {
1909:       int a = que.front();
1910:       que.pop();
1911:       for(auto &b : graph[a]) {
1912:         int c = match[b];
1913:         if(c >= 0 && dist[c] == -1) {
1914:           dist[c] = dist[a] + 1;
1915:           que.emplace(c);
1916:         }
1917:       }
1918:     }
1919:   }
1920:
1921:   bool dfs(int a) {
1922:     vv[a] = true;
1923:     for(auto &b : graph[a]) {
1924:       int c = match[b];
1925:       if(c < 0 || (!vv[c] && dist[c] == dist[a] + 1 && dfs(c))) {
1926:         match[b] = a;
1927:         used[a] = true;
1928:         return (true);
1929:       }
1930:     }
1931:     return (false);
1932:   }
```

```
1933:
1934:   int bipartite_matching() {
1935:     int ret = 0;
1936:     while(true) {
1937:       bfs();
1938:       vv.assign(graph.size(), false);
1939:       int flow = 0;
1940:       for(int i = 0; i < graph.size(); i++) {
1941:         if(!used[i] && dfs(i)) ++flow;
1942:       }
1943:       if(flow == 0) return (ret);
1944:       ret += flow;
1945:     }
1946:   }
1947:
1948:   void output() {
1949:     for(int i = 0; i < match.size(); i++) {
1950:       if(~match[i]) {
1951:         cout << match[i] << "-" << i << endl;
1952:       }
1953:     }
1954:   }
1955: };
1956:
1957:
1958: ######################################################
1959: ################## lowlink.cpp ###################
1960: ######################################################
1961:
1962: template< typename G >
1963: struct LowLink {
1964:   const G &g;
1965:   vector< int > used, ord, low;
1966:   vector< int > articulation;
1967:   vector< pair< int, int > > bridge;
1968:
1969:   LowLink(const G &g) : g(g) {}
1970:
1971:   int dfs(int idx, int k, int par) {
1972:     used[idx] = true;
1973:     ord[idx] = k++;
1974:     low[idx] = ord[idx];
1975:     bool is_articulation = false;
1976:     int cnt = 0;
1977:     for(auto &to : g[idx]) {
1978:       if(!used[to]) {
1979:         ++cnt;
1980:         k = dfs(to, k, idx);
1981:         low[idx] = min(low[idx], low[to]);
1982:         is_articulation |= ~par && low[to] >= ord[idx];
1983:         if(ord[idx] < low[to]) bridge.emplace_back(minmax(idx, (int) to));
1984:       } else if(to != par) {
1985:         low[idx] = min(low[idx], ord[to]);
1986:       }
1987:     }
1988:     is_articulation |= par == -1 && cnt > 1;
1989:     if(is_articulation) articulation.push_back(idx);
1990:     return k;
1991:   }
1992:
1993:   virtual void build() {
1994:     used.assign(g.size(), 0);
1995:     ord.assign(g.size(), 0);
1996:     low.assign(g.size(), 0);
1997:     int k = 0;
1998:     for(int i = 0; i < g.size(); i++) {
1999:       if(!used[i]) k = dfs(i, k, -1);
2000:     }
2001:   }
2002: };
```

```
2003:
2004:
2005: ####################################################
2006: ############### maximum-clique.cpp ################
2007: ####################################################
2008:
2009: template< typename T >
2010: T maximum_clique(Matrix< bool > g, function< T(vector< int >) > f) {
2011:
2012:   int N = (int) g.size(), M = 0;
2013:   vector< int > deg(N), v(N);
2014:   for(int i = 0; i < N; i++) {
2015:     for(int j = 0; j < i; j++) {
2016:       assert(g[i][j] == g[j][i]);
2017:       if(g[i][j]) {
2018:         ++deg[i];
2019:         ++M;
2020:       }
2021:     }
2022:   }
2023:   T t = 0;
2024:   int lim = (int) sqrt(2 * M);
2025:
2026:   for(int i = 0; i < N; i++) {
2027:     vector< int > notice;
2028:     for(int j = 0; j < N; j++) {
2029:       if(!v[j] && deg[j] < lim) {
2030:         for(int k = 0; k < N; k++) {
2031:           if(j == k) continue;
2032:           if(g[j][k]) notice.emplace_back(k);
2033:         }
2034:         notice.emplace_back(j);
2035:         break;
2036:       }
2037:     }
2038:     if(notice.empty()) break;
2039:     int neighbor = (int) notice.size() - 1;
2040:     vector< int > bit(neighbor);
2041:     for(int j = 0; j < neighbor; j++) {
2042:       for(int k = 0; k < j; k++) {
2043:         if(!g[notice[j]][notice[k]]) {
2044:           bit[j] |= 1 << k;
2045:           bit[k] |= 1 << j;
2046:         }
2047:       }
2048:     }
2049:     for(int j = 0; j < (1 << neighbor); j++) {
2050:       bool ok = true;
2051:       for(int k = 0; k < neighbor; k++) {
2052:         if((j >> k) & 1) ok &= (j & bit[k]) == 0;
2053:       }
2054:       if(ok) {
2055:         vector< int > stock{notice.back()};
2056:         for(int k = 0; k < neighbor; k++) {
2057:           if((j >> k) & 1) stock.emplace_back(notice[k]);
2058:         }
2059:         t = max(t, f(stock));
2060:       }
2061:     }
2062:     v[notice.back()] = true;
2063:     for(int j = 0; j < N; j++) {
2064:       if(g[j][notice.back()]) {
2065:         --deg[j];
2066:         g[notice.back()][j] = g[j][notice.back()] = false;
2067:       }
2068:     }
2069:   }
2070:
2071:   vector< int > notice;
2072:   for(int j = 0; j < N; j++) {
```

```
2073:     if(!v[j]) notice.emplace_back(j);
2074:   }
2075:   int neighbor = (int) notice.size();
2076:   vector< int > bit(neighbor);
2077:   for(int j = 0; j < neighbor; j++) {
2078:     for(int k = 0; k < j; k++) {
2079:       if(!g[notice[j]][notice[k]]) {
2080:         bit[j] |= 1 << k;
2081:         bit[k] |= 1 << j;
2082:       }
2083:     }
2084:   }
2085:   for(int j = 0; j < (1 << neighbor); j++) {
2086:     bool ok = true;
2087:     for(int k = 0; k < neighbor; k++) {
2088:       if((j >> k) & 1) ok &= (j & bit[k]) == 0;
2089:     }
2090:     if(ok) {
2091:       vector< int > stock;
2092:       for(int k = 0; k < neighbor; k++) {
2093:         if((j >> k) & 1) stock.emplace_back(notice[k]);
2094:       }
2095:       t = max(t, f(stock));
2096:     }
2097:   }
2098:   return t;
2099: }
2100:
2101:
2102:
2103: ------------------------------------------------
2104: |                     other                    |
2105: ------------------------------------------------
2106:
2107: ################################################
2108: ##################### mo.cpp ###################
2109: ################################################
2110:
2111: struct Mo {
2112:   using ADD = function< void(int) >;
2113:   using DEL = function< void(int) >;
2114:   using REM = function< void(int) >;
2115:
2116:   int width;
2117:   vector< int > left, right, order;
2118:   vector< bool > v;
2119:
2120:   Mo(int N, int Q) : width((int) sqrt(N)), order(Q), v(N) {
2121:     iota(begin(order), end(order), 0);
2122:   }
2123:
2124:   void add(int l, int r) { /* [l, r) */
2125:     left.emplace_back(l);
2126:     right.emplace_back(r);
2127:   }
2128:
2129:   int run(const ADD &add, const DEL &del, const REM &rem) {
2130:     assert(left.size() == order.size());
2131:     sort(begin(order), end(order), [&](int a, int b) {
2132:       int ablock = left[a] / width, bblock = left[b] / width;
2133:       if(ablock != bblock) return ablock < bblock;
2134:       if(ablock & 1) return right[a] < right[b];
2135:       return right[a] > right[b];
2136:     });
2137:     int nl = 0, nr = 0;
2138:     auto push = [&](int idx) {
2139:       v[idx].flip();
2140:       if(v[idx]) add(idx);
2141:       else del(idx);
2142:     };
```

```
2143:     for(auto idx : order) {
2144:       while(nl > left[idx]) push(--nl);
2145:       while(nr < right[idx]) push(nr++);
2146:       while(nl < left[idx]) push(nl++);
2147:       while(nr > right[idx]) push(--nr);
2148:       rem(idx);
2149:     }
2150:   }
2151: };
2152:
2153:
2154: ######################################################
2155: ##################### dice.cpp #######################
2156: ######################################################
2157:
2158: struct Dice
2159: {
2160:   // int x, y;
2161:   int l, r, f, b, d, u;
2162:
2163:   void RollN()
2164:   {
2165:     //  --y;
2166:     int buff = d;
2167:     d = f;
2168:     f = u;
2169:     u = b;
2170:     b = buff;
2171:   }
2172:
2173:   void RollS()
2174:   {
2175:     // ++y;
2176:     int buff = d;
2177:     d = b;
2178:     b = u;
2179:     u = f;
2180:     f = buff;
2181:   }
2182:
2183:   void RollL() // ----->
2184:   {
2185:     int buff = f;
2186:     f = l;
2187:     l = b;
2188:     b = r;
2189:     r = buff;
2190:   }
2191:
2192:   void RollR() // <------
2193:   {
2194:     int buff = f;
2195:     f = r;
2196:     r = b;
2197:     b = l;
2198:     l = buff;
2199:   }
2200:
2201:   void RollE() // .o -> o.
2202:   {
2203:     // --x;
2204:     int buff = d;
2205:     d = l;
2206:     l = u;
2207:     u = r;
2208:     r = buff;
2209:   }
2210:
2211:
2212:   void RollW() // o. -> .o
```

```
2213:   {
2214:     //  ++x;
2215:     int buff = d;
2216:     d = r;
2217:     r = u;
2218:     u = l;
2219:     l = buff;
2220:   }
2221:
2222:
2223:   vector< Dice > makeDice()
2224:   {
2225:     vector< Dice > ret;
2226:     for(int i = 0; i < 6; i++) {
2227:       Dice d(*this);
2228:       if(i == 1) d.RollN();
2229:       if(i == 2) d.RollS();
2230:       if(i == 3) d.RollS(), d.RollS();
2231:       if(i == 4) d.RollL();
2232:       if(i == 5) d.RollR();
2233:       for(int j = 0; j < 4; j++) {
2234:         ret.emplace_back(d);
2235:         d.RollE();
2236:       }
2237:     }
2238:     return (ret);
2239:   }
2240: };
2241:
2242:
2243: ######################################################
2244: ################### timer.cpp ########################
2245: ######################################################
2246:
2247: struct Timer {
2248:   chrono::high_resolution_clock::time_point st;
2249:
2250:   Timer() { reset(); }
2251:
2252:   void reset() {
2253:     st = chrono::high_resolution_clock::now();
2254:   }
2255:
2256:   chrono::milliseconds::rep elapsed() {
2257:     auto ed = chrono::high_resolution_clock::now();
2258:     return chrono::duration_cast< chrono::milliseconds >(ed - st).count();
2259:   }
2260: };
2261:
2262:
2263: ######################################################
2264: ################# fast-input.cpp #####################
2265: ######################################################
2266:
2267: template< int sz >
2268: struct FastInput {
2269:   char buf[sz + 1];
2270:   char *o;
2271:
2272:   FastInput() { init(); }
2273:
2274:   void init() {
2275:     o = buf;
2276:     buf[fread(buf, sizeof(char), sizeof(char) * sz, stdin)] = '\0';
2277:   }
2278:
2279:   int64_t read() {
2280:     int64_t ret = 0, sign = 1;
2281:     while(*o && *o <= 32) ++o;
2282:     if(*o == '-') sign *= -1, ++o;
```

```
2283:     while(*o >= '0' && *o <= '9') {
2284:       ret *= 10;
2285:       ret += *o++ - '0';
2286:     }
2287:     return ret * sign;
2288:   }
2289: };
2290:
2291:
2292: ######################################################
2293: ######## offline-dynamic-connectivity.cpp ########
2294: ######################################################
2295:
2296: struct OfflineDynamicConnectivity {
2297:   using edge = pair< int, int >;
2298:
2299:   UnionFindUndo uf;
2300:   int V, Q, segsz;
2301:   vector< vector< edge > > seg;
2302:   int comp;
2303:
2304:   vector< pair< pair< int, int >, edge > > pend;
2305:   map< edge, int > cnt, appear;
2306:
2307:   OfflineDynamicConnectivity(int V, int Q) : uf(V), V(V), Q(Q), comp(V) {
2308:     segsz = 1;
2309:     while(segsz < Q) segsz <<= 1;
2310:     seg.resize(2 * segsz - 1);
2311:   }
2312:
2313:   void insert(int idx, int s, int t) {
2314:     auto e = minmax(s, t);
2315:     if(cnt[e]++ == 0) appear[e] = idx;
2316:   }
2317:
2318:   void erase(int idx, int s, int t) {
2319:     auto e = minmax(s, t);
2320:     if(--cnt[e] == 0) pend.emplace_back(make_pair(appear[e], idx), e);
2321:   }
2322:
2323:   void add(int a, int b, const edge &e, int k, int l, int r) {
2324:     if(r <= a || b <= l) return;
2325:     if(a <= l && r <= b) {
2326:       seg[k].emplace_back(e);
2327:       return;
2328:     }
2329:     add(a, b, e, 2 * k + 1, l, (l + r) >> 1);
2330:     add(a, b, e, 2 * k + 2, (l + r) >> 1, r);
2331:   }
2332:
2333:   void add(int a, int b, const edge &e) {
2334:     add(a, b, e, 0, 0, segsz);
2335:   }
2336:
2337:   void build() {
2338:     for(auto &p : cnt) {
2339:       if(p.second > 0) pend.emplace_back(make_pair(appear[p.first], Q), p.first);
2340:     }
2341:     for(auto &s : pend) {
2342:       add(s.first.first, s.first.second, s.second);
2343:     }
2344:   }
2345:
2346:   int run(const function< void(int) > &f, int k = 0) {
2347:     int add = 0;
2348:     for(auto &e : seg[k]) {
2349:       add += uf.unite(e.first, e.second);
2350:     }
2351:     comp -= add;
2352:     if(k < segsz - 1) {
```

```
 2353:       run(f, 2 * k + 1);
 2354:       run(f, 2 * k + 2);
 2355:     } else if(k - (segsz - 1) < Q) {
 2356:       int query_index = k - (segsz - 1);
 2357:       f(query_index);
 2358:     }
 2359:     for(auto &e : seg[k]) {
 2360:       uf.undo();
 2361:     }
 2362:     comp += add;
 2363:   }
 2364: };
 2365:
 2366:
 2367: ##################################################
 2368: ########## random-number-generator.cpp ############
 2369: ##################################################
 2370:
 2371: struct RandomNumberGenerator {
 2372:   mt19937 mt;
 2373:
 2374:   RandomNumberGenerator() : mt(chrono::steady_clock::now().time_since_epoch().coun
t()) {}
 2375:
 2376:   int operator()(int a, int b) { // [a, b)
 2377:     uniform_int_distribution< int > dist(a, b - 1);
 2378:     return dist(mt);
 2379:   }
 2380:
 2381:   int operator()(int b) { // [0, b)
 2382:     return (*this)(0, b);
 2383:   }
 2384: };
 2385:
 2386:
 2387: ##################################################
 2388: ################ mo-rollback.cpp #################
 2389: ##################################################
 2390:
 2391: struct MoRollBack {
 2392:   using ADD = function< void(int) >;
 2393:   using REM = function< void(int) >;
 2394:   using RESET = function< void() >;
 2395:   using SNAPSHOT = function< void() >;
 2396:   using ROLLBACK = function< void() >;
 2397:
 2398:   int width;
 2399:   vector< int > left, right, order;
 2400:
 2401:   MoRollBack(int N, int Q) : width((int) sqrt(N)), order(Q) {
 2402:     iota(begin(order), end(order), 0);
 2403:   }
 2404:
 2405:   void add(int l, int r) { /* [l, r) */
 2406:     left.emplace_back(l);
 2407:     right.emplace_back(r);
 2408:   }
 2409:
 2410:   int run(const ADD &add, const REM &rem, const RESET &reset, const SNAPSHOT &snap
shot, const ROLLBACK &rollback) {
 2411:     assert(left.size() == order.size());
 2412:     sort(begin(order), end(order), [&](int a, int b) {
 2413:       int ablock = left[a] / width, bblock = left[b] / width;
 2414:       if(ablock != bblock) return ablock < bblock;
 2415:       return right[a] < right[b];
 2416:     });
 2417:     reset();
 2418:     for(auto idx : order) {
 2419:       if(right[idx] - left[idx] < width) {
 2420:         for(int i = left[idx]; i < right[idx]; i++) add(i);
```

```
2421:         rem(idx);
2422:         rollback();
2423:       }
2424:     }
2425:     int nr = 0, last_block = -1;
2426:     for(auto idx : order) {
2427:       if(right[idx] - left[idx] < width) continue;
2428:       int block = left[idx] / width;
2429:       if(last_block != block) {
2430:         reset();
2431:         last_block = block;
2432:         nr = (block + 1) * width;
2433:       }
2434:       while(nr < right[idx]) add(nr++);
2435:       snapshot();
2436:       for(int j = (block + 1) * width - 1; j >= left[idx]; j--) add(j);
2437:       rem(idx);
2438:       rollback();
2439:     }
2440:   }
2441: };
2442:
2443:
2444:
2445: ######################################################
2446: ################## compress.cpp ######################
2447: ######################################################
2448:
2449: template< typename T >
2450: struct Compress {
2451:   vector< T > xs;
2452:
2453:   Compress() = default;
2454:
2455:   Compress(const vector< T > &vs) {
2456:     add(vs);
2457:   }
2458:
2459:   Compress(const initializer_list< vector< T > > &vs) {
2460:     for(auto &p : vs) add(p);
2461:   }
2462:
2463:   void add(const vector< T > &vs) {
2464:     copy(begin(vs), end(vs), back_inserter(xs));
2465:   }
2466:
2467:   void add(const T &x) {
2468:     xs.emplace_back(x);
2469:   }
2470:
2471:   void build() {
2472:     sort(begin(xs), end(xs));
2473:     xs.erase(unique(begin(xs), end(xs)), end(xs));
2474:   }
2475:
2476:   vector< int > get(const vector< T > &vs) const {
2477:     vector< int > ret;
2478:     transform(begin(vs), end(vs), back_inserter(ret), [&](const T &x) {
2479:       return lower_bound(begin(xs), end(xs), x) - begin(xs);
2480:     });
2481:     return ret;
2482:   }
2483:
2484:   int get(const T &x) const {
2485:     return lower_bound(begin(xs), end(xs), x) - begin(xs);
2486:   }
2487:
2488:   const T &operator[](int k) const {
2489:     return xs[k];
2490:   }
```

```
2491: };
2492:
2493:
2494: ------------------------------------------------
2495: |                   template                   |
2496: ------------------------------------------------
2497:
2498: ####################################################
2499: ################## template.cpp ####################
2500: ####################################################
2501:
2502: #include<bits/stdc++.h>
2503:
2504: using namespace std;
2505:
2506: using int64 = long long;
2507: const int mod = 1e9 + 7;
2508:
2509: const int64 infll = (1LL << 62) - 1;
2510: const int inf = (1 << 30) - 1;
2511:
2512: struct IoSetup {
2513:   IoSetup() {
2514:     cin.tie(nullptr);
2515:     ios::sync_with_stdio(false);
2516:     cout << fixed << setprecision(10);
2517:     cerr << fixed << setprecision(10);
2518:   }
2519: } iosetup;
2520:
2521:
2522: template< typename T1, typename T2 >
2523: ostream &operator<<(ostream &os, const pair< T1, T2 >& p) {
2524:   os << p.first << " " << p.second;
2525:   return os;
2526: }
2527:
2528: template< typename T1, typename T2 >
2529: istream &operator>>(istream &is, pair< T1, T2 > &p) {
2530:   is >> p.first >> p.second;
2531:   return is;
2532: }
2533:
2534: template< typename T >
2535: ostream &operator<<(ostream &os, const vector< T > &v) {
2536:   for(int i = 0; i < (int) v.size(); i++) {
2537:     os << v[i] << (i + 1 != v.size() ? " " : "");
2538:   }
2539:   return os;
2540: }
2541:
2542: template< typename T >
2543: istream &operator>>(istream &is, vector< T > &v) {
2544:   for(T &in : v) is >> in;
2545:   return is;
2546: }
2547:
2548: template< typename T1, typename T2 >
2549: inline bool chmax(T1 &a, T2 b) { return a < b && (a = b, true); }
2550:
2551: template< typename T1, typename T2 >
2552: inline bool chmin(T1 &a, T2 b) { return a > b && (a = b, true); }
2553:
2554: template< typename T = int64 >
2555: vector< T > make_v(size_t a) {
2556:   return vector< T >(a);
2557: }
2558:
2559: template< typename T, typename... Ts >
2560: auto make_v(size_t a, Ts... ts) {
```

```
2561:   return vector< decltype(make_v< T >(ts...)) >(a, make_v< T >(ts...));
2562: }
2563:
2564: template< typename T, typename V >
2565: typename enable_if< is_class< T >::value == 0 >::type fill_v(T &t, const V &v) {
2566:   t = v;
2567: }
2568:
2569: template< typename T, typename V >
2570: typename enable_if< is_class< T >::value != 0 >::type fill_v(T &t, const V &v) {
2571:   for(auto &e : t) fill_v(e, v);
2572: }
2573:
2574: template< typename F >
2575: struct FixPoint : F {
2576:   FixPoint(F &&f) : F(forward< F >(f)) {}
2577:
2578:   template< typename... Args >
2579:   decltype(auto) operator()(Args &&... args) const {
2580:     return F::operator()(*this, forward< Args >(args)...);
2581:   }
2582: };
2583:
2584: template< typename F >
2585: inline decltype(auto) MFP(F &&f) {
2586:   return FixPoint< F >{forward< F >(f)};
2587: }
2588:
2589:
2590: ------------------------------------------------
2591: |                      math                    |
2592: ------------------------------------------------
2593:
2594: ################################################
2595: ################ prime-factor.cpp ###############
2596: ################################################
2597:
2598: map< int64_t, int > prime_factor(int64_t n) {
2599:   map< int64_t, int > ret;
2600:   for(int64_t i = 2; i * i <= n; i++) {
2601:     while(n % i == 0) {
2602:       ret[i]++;
2603:       n /= i;
2604:     }
2605:   }
2606:   if(n != 1) ret[n] = 1;
2607:   return ret;
2608: }
2609:
2610:
2611: ################################################
2612: ########### stirling-number-second.cpp ##########
2613: ################################################
2614:
2615: template< typename T >
2616: T stirling_number_second(int n, int k) {
2617:   Combination< T > table(k);
2618:   T ret = 0;
2619:   for(int i = 0; i <= k; i++) {
2620:     auto add = T(i).pow(n) * table.C(k, i);
2621:     if((k - i) & 1) ret -= add;
2622:     else ret += add;
2623:   }
2624:   return ret * table.rfact(k);
2625: }
2626:
2627:
2628:
2629: ################################################
2630: ################## mod-int.cpp ##################
```

```
2631: ##################################################
2632:
2633: template< int mod >
2634: struct ModInt {
2635:   int x;
2636:
2637:   ModInt() : x(0) {}
2638:
2639:   ModInt(int64_t y) : x(y >= 0 ? y % mod : (mod - (-y) % mod) % mod) {}
2640:
2641:   ModInt &operator+=(const ModInt &p) {
2642:     if((x += p.x) >= mod) x -= mod;
2643:     return *this;
2644:   }
2645:
2646:   ModInt &operator-=(const ModInt &p) {
2647:     if((x += mod - p.x) >= mod) x -= mod;
2648:     return *this;
2649:   }
2650:
2651:   ModInt &operator*=(const ModInt &p) {
2652:     x = (int) (1LL * x * p.x % mod);
2653:     return *this;
2654:   }
2655:
2656:   ModInt &operator/=(const ModInt &p) {
2657:     *this *= p.inverse();
2658:     return *this;
2659:   }
2660:
2661:   ModInt operator-() const { return ModInt(-x); }
2662:
2663:   ModInt operator+(const ModInt &p) const { return ModInt(*this) += p; }
2664:
2665:   ModInt operator-(const ModInt &p) const { return ModInt(*this) -= p; }
2666:
2667:   ModInt operator*(const ModInt &p) const { return ModInt(*this) *= p; }
2668:
2669:   ModInt operator/(const ModInt &p) const { return ModInt(*this) /= p; }
2670:
2671:   bool operator==(const ModInt &p) const { return x == p.x; }
2672:
2673:   bool operator!=(const ModInt &p) const { return x != p.x; }
2674:
2675:   ModInt inverse() const {
2676:     int a = x, b = mod, u = 1, v = 0, t;
2677:     while(b > 0) {
2678:       t = a / b;
2679:       swap(a -= t * b, b);
2680:       swap(u -= t * v, v);
2681:     }
2682:     return ModInt(u);
2683:   }
2684:
2685:   ModInt pow(int64_t n) const {
2686:     ModInt ret(1), mul(x);
2687:     while(n > 0) {
2688:       if(n & 1) ret *= mul;
2689:       mul *= mul;
2690:       n >>= 1;
2691:     }
2692:     return ret;
2693:   }
2694:
2695:   friend ostream &operator<<(ostream &os, const ModInt &p) {
2696:     return os << p.x;
2697:   }
2698:
2699:   friend istream &operator>>(istream &is, ModInt &a) {
2700:     int64_t t;
```

```
2701:     is >> t;
2702:     a = ModInt< mod >(t);
2703:     return (is);
2704:   }
2705:
2706:   static int get_mod() { return mod; }
2707: };
2708:
2709: using modint = ModInt< mod >;
2710:
2711:
2712: ######################################################
2713: ################## binomial.cpp ######################
2714: ######################################################
2715:
2716: template< typename T >
2717: T binomial(int64_t N, int64_t K) {
2718:   if(K < 0 || N < K) return 0;
2719:   T ret = 1;
2720:   for(T i = 1; i <= K; ++i) {
2721:     ret *= N--;
2722:     ret /= i;
2723:   }
2724:   return ret;
2725: }
2726:
2727:
2728: ######################################################
2729: ################## factorial.cpp #####################
2730: ######################################################
2731:
2732: template< typename T >
2733: T factorial(int64_t n) {
2734:   if(n >= T::get_mod()) return 0;
2735:   if(n == 0) return 1;
2736:
2737:   const int64_t sn = sqrt(n);
2738:   const T sn_inv = T(1) / sn;
2739:
2740:   Combination< modint > comb(sn);
2741:   using P = vector< T >;
2742:
2743:   ArbitraryModConvolution< T > fft;
2744:   using FPS = FormalPowerSeries< T >;
2745:   auto mult = [&](const typename FPS::P &a, const typename FPS::P &b) {
2746:     auto ret = fft.multiply(a, b);
2747:     return typename FPS::P(ret.begin(), ret.end());
2748:   };
2749:   FPS::set_fft(mult);
2750:
2751:
2752:   auto shift = [&](const P &f, T dx) {
2753:     int n = (int) f.size();
2754:     T a = dx * sn_inv;
2755:     auto p1 = P(f);
2756:     for(int i = 0; i < n; i++) {
2757:       T d = comb.rfact(i) * comb.rfact((n - 1) - i);
2758:       if(((n - 1 - i) & 1)) d = -d;
2759:       p1[i] *= d;
2760:     }
2761:     auto p2 = P(2 * n);
2762:     for(int i = 0; i < p2.size(); i++) {
2763:       p2[i] = (a.x + i - n) <= 0 ? 1 : a + i - n;
2764:     }
2765:     for(int i = 1; i < p2.size(); i++) {
2766:       p2[i] *= p2[i - 1];
2767:     }
2768:     T prod = p2[2 * n - 1];
2769:     T prod_inv = T(1) / prod;
2770:     for(int i = 2 * n - 1; i > 0; --i) {
```

```
2771:       p2[i] = prod_inv * p2[i - 1];
2772:       prod_inv *= a + i - n;
2773:     }
2774:     p2[0] = prod_inv;
2775:     auto p3 = fft.multiply(p1, p2, (int) p2.size());
2776:     p1 = P(p3.begin() + p1.size(), p3.begin() + p2.size());
2777:     prod = 1;
2778:     for(int i = 0; i < n; i++) {
2779:       prod *= a + n - 1 - i;
2780:     }
2781:     for(int i = n - 1; i >= 0; --i) {
2782:       p1[i] *= prod;
2783:       prod *= p2[n + i] * (a + i - n);
2784:     }
2785:     return p1;
2786:   };
2787:   function< P(int) > rec = [&](int64_t n) {
2788:     if(n == 1) return P({1, 1 + sn});
2789:     int64_t nh = n >> 1;
2790:     auto a1 = rec(nh);
2791:     auto a2 = shift(a1, nh);
2792:     auto b1 = shift(a1, sn * nh);
2793:     auto b2 = shift(a1, sn * nh + nh);
2794:     for(int i = 0; i <= nh; i++) a1[i] *= a2[i];
2795:     for(int i = 1; i <= nh; i++) a1.emplace_back(b1[i] * b2[i]);
2796:     if(n & 1) {
2797:       for(int64_t i = 0; i < n; i++) {
2798:         a1[i] *= n + 1LL * sn * i;
2799:       }
2800:       T prod = 1;
2801:       for(int64_t i = 1LL * n * sn; i < 1LL * n * sn + n; i++) {
2802:         prod *= (i + 1);
2803:       }
2804:       a1.push_back(prod);
2805:     }
2806:     return a1;
2807:   };
2808:   auto vs = rec(sn);
2809:   T ret = 1;
2810:   for(int64_t i = 0; i < sn; i++) ret *= vs[i];
2811:   for(int64_t i = 1LL * sn * sn + 1; i <= n; i++) ret *= i;
2812:   return ret;
2813: }
2814:
2815:
2816: ////////////////////////////////////////////////
2817: /////////////// binomial-table.cpp ///////////////
2818: ////////////////////////////////////////////////
2819:
2820: template< typename T >
2821: vector< vector< T > > binomial_table(int N) {
2822:   vector< vector< T > > mat(N + 1, vector< T >(N + 1));
2823:   for(int i = 0; i <= N; i++) {
2824:     for(int j = 0; j <= i; j++) {
2825:       if(j == 0 || j == i) mat[i][j] = 1;
2826:       else mat[i][j] = mat[i - 1][j - 1] + mat[i - 1][j];
2827:     }
2828:   }
2829:   return mat;
2830: }
2831:
2832:
2833: ////////////////////////////////////////////////
2834: ////////// polynomial-interpolation.cpp //////////
2835: ////////////////////////////////////////////////
2836:
2837: template< class T >
2838: FormalPowerSeries< T > polynomial_interpolation(const FormalPowerSeries< T > &xs,
const vector< T > &ys) {
2839:   assert(xs.size() == ys.size());
```

```
 2840:   using FPS = FormalPowerSeries< T >;
 2841:   PolyBuf< T > buf(xs);
 2842:   FPS w = buf.query(0, xs.size()).diff();
 2843:   auto vs = multipoint_evaluation(w, xs, buf);
 2844:   function< FPS(int, int) > rec = [&](int l, int r) -> FPS {
 2845:     if(r - l == 1) return {ys[l] / vs[l]};
 2846:     int m = (l + r) >> 1;
 2847:     return rec(l, m) * buf.query(m, r) + rec(m, r) * buf.query(l, m);
 2848:   };
 2849:   return rec(0, xs.size());
 2850: }
 2851:
 2852:
 2853: ######################################################
 2854: ################ bell-number.cpp ##################
 2855: ######################################################
 2856:
 2857: template< typename T >
 2858: T bell_number(int n, int k) {
 2859:   if(n == 0) return 1;
 2860:   k = min(k, n);
 2861:   Combination< T > uku(k);
 2862:   T ret = 0;
 2863:   vector< T > pref(k + 1);
 2864:   pref[0] = 1;
 2865:   for(int i = 1; i <= k; i++) {
 2866:     if(i & 1) pref[i] = pref[i - 1] - uku.rfact(i);
 2867:     else pref[i] = pref[i - 1] + uku.rfact(i);
 2868:   }
 2869:   for(int i = 1; i <= k; i++) {
 2870:     ret += T(i).pow(n) * uku.rfact(i) * pref[k - i];
 2871:   }
 2872:   return ret;
 2873: }
 2874:
 2875:
 2876: ######################################################
 2877: ################## divisor.cpp ###################
 2878: ######################################################
 2879:
 2880: vector< int64_t > divisor(int64_t n) {
 2881:   vector< int64_t > ret;
 2882:   for(int64_t i = 1; i * i <= n; i++) {
 2883:     if(n % i == 0) {
 2884:       ret.push_back(i);
 2885:       if(i * i != n) ret.push_back(n / i);
 2886:     }
 2887:   }
 2888:   sort(begin(ret), end(ret));
 2889:   return (ret);
 2890: }
 2891:
 2892:
 2893: ######################################################
 2894: ####### arbitrary-mod-convolution-long.cpp #######
 2895: ######################################################
 2896:
 2897: template< typename T >
 2898: struct ArbitraryModConvolutionLong {
 2899:   using real = FastFourierTransform::real;
 2900:   using C = FastFourierTransform::C;
 2901:
 2902:   ArbitraryModConvolutionLong() = default;
 2903:
 2904:   vector< T > multiply(const vector< T > &a, const vector< T > &b, int need = -1)
{
 2905:     if(need == -1) need = a.size() + b.size() - 1;
 2906:     int nbase = 0;
 2907:     while((1 << nbase) < need) nbase++;
 2908:     FastFourierTransform::ensure_base(nbase);
```

```
2909:     int sz = 1 << nbase;
2910:     vector< C > fa(sz);
2911:     for(int i = 0; i < a.size(); i++) {
2912:       fa[i] = C(a[i].x & ((1 << 19) - 1), a[i].x >> 19);
2913:     }
2914:     fft(fa, sz);
2915:     vector< C > fb(sz);
2916:     if(a == b) {
2917:       fb = fa;
2918:     } else {
2919:       for(int i = 0; i < b.size(); i++) {
2920:         fb[i] = C(b[i].x & ((1 << 19) - 1), b[i].x >> 19);
2921:       }
2922:       fft(fb, sz);
2923:     }
2924:     real ratio = 0.25 / sz;
2925:     C r2(0, -1), r3(ratio, 0), r4(0, -ratio), r5(0, 1);
2926:     for(int i = 0; i <= (sz >> 1); i++) {
2927:       int j = (sz - i) & (sz - 1);
2928:       C a1 = (fa[i] + fa[j].conj());
2929:       C a2 = (fa[i] - fa[j].conj()) * r2;
2930:       C b1 = (fb[i] + fb[j].conj()) * r3;
2931:       C b2 = (fb[i] - fb[j].conj()) * r4;
2932:       if(i != j) {
2933:         C c1 = (fa[j] + fa[i].conj());
2934:         C c2 = (fa[j] - fa[i].conj()) * r2;
2935:         C d1 = (fb[j] + fb[i].conj()) * r3;
2936:         C d2 = (fb[j] - fb[i].conj()) * r4;
2937:         fa[i] = c1 * d1 + c2 * d2 * r5;
2938:         fb[i] = c1 * d2 + c2 * d1;
2939:       }
2940:       fa[j] = a1 * b1 + a2 * b2 * r5;
2941:       fb[j] = a1 * b2 + a2 * b1;
2942:     }
2943:     fft(fa, sz);
2944:     fft(fb, sz);
2945:     vector< T > ret(need);
2946:     auto mul1 = T(2).pow(19);
2947:     auto mul2 = T(2).pow(38);
2948:     for(int i = 0; i < need; i++) {
2949:       int64_t aa = llround(fa[i].x);
2950:       int64_t bb = llround(fb[i].x);
2951:       int64_t cc = llround(fa[i].y);
2952:       aa = T(aa).x, bb = T(bb).x, cc = T(cc).x;
2953:       ret[i] = (mul1 * bb) + (mul2 * cc) + aa;
2954:     }
2955:     return ret;
2956:   }
2957: };
2958:
2959:
2960: ////////////////////////////////////////////////////
2961: ////////////////// mod-pow.cpp /////////////////////
2962: ////////////////////////////////////////////////////
2963:
2964: template< typename T >
2965: T mod_pow(T x, T n, const T &p) {
2966:   T ret = 1;
2967:   while(n > 0) {
2968:     if(n & 1) (ret *= x) %= p;
2969:     (x *= x) %= p;
2970:     n >>= 1;
2971:   }
2972:   return ret;
2973: }
2974:
2975:
2976:
2977: ////////////////////////////////////////////////////
2978: ///////////////// prime-table.cpp //////////////////
```

```
2979: ##################################################
2980:
2981: vector< bool > prime_table(int n) {
2982:   vector< bool > prime(n + 1, true);
2983:   if(n >= 0) prime[0] = false;
2984:   if(n >= 1) prime[1] = false;
2985:   for(int i = 2; i * i <= n; i++) {
2986:     if(!prime[i]) continue;
2987:     for(int j = i + i; j <= n; j += i) {
2988:       prime[j] = false;
2989:     }
2990:   }
2991:   return prime;
2992: }
2993:
2994:
2995: ##################################################
2996: ########## multipoint-evaluation.cpp ############
2997: ##################################################
2998:
2999: template< typename T >
3000: struct PolyBuf {
3001:   using FPS = FormalPowerSeries< T >;
3002:   const FPS xs;
3003:   using pi = pair< int, int >;
3004:   map< pi, FPS > buf;
3005:
3006:   PolyBuf(const FPS &xs) : xs(xs) {}
3007:
3008:   const FPS &query(int l, int r) {
3009:     if(buf.count({l, r})) return buf[{l, r}];
3010:     if(l + 1 == r) return buf[{l, r}] = {-xs[l], 1};
3011:     return buf[{l, r}] = query(l, (l + r) >> 1) * query((l + r) >> 1, r);
3012:   }
3013: };
3014:
3015:
3016: template< typename T >
3017: FormalPowerSeries< T > multipoint_evaluation(const FormalPowerSeries< T > &as, con
st FormalPowerSeries< T > &xs, PolyBuf< T > &buf) {
3018:   using FPS = FormalPowerSeries< T >;
3019:   FPS ret;
3020:   const int B = 64;
3021:   function< void(FPS, int, int) > rec = [&](FPS a, int l, int r) -> void {
3022:     a %= buf.query(l, r);
3023:     if(a.size() <= B) {
3024:       for(int i = l; i < r; i++) ret.emplace_back(a.eval(xs[i]));
3025:       return;
3026:     }
3027:     rec(a, l, (l + r) >> 1);
3028:     rec(a, (l + r) >> 1, r);
3029:   };
3030:   rec(as, 0, xs.size());
3031:   return ret;
3032: };
3033:
3034: template< typename T >
3035: FormalPowerSeries< T > multipoint_evaluation(const FormalPowerSeries< T > &as, con
st FormalPowerSeries< T > &xs) {
3036:   PolyBuf< T > buff(xs);
3037:   return multipoint_evaluation(as, xs, buff);
3038: }
3039:
3040:
3041:
3042: ##################################################
3043: ################## mod-sqrt.cpp ##################
3044: ##################################################
3045:
3046: template< typename T >
```

```
3047: T mod_sqrt(const T &a, const T &p) {
3048:   if(a == 0) return 0;
3049:   if(p == 2) return a;
3050:   if(mod_pow(a, (p - 1) >> 1, p) != 1) return -1;
3051:   T b = 1;
3052:   while(mod_pow(b, (p - 1) >> 1, p) == 1) ++b;
3053:   T e = 0, m = p - 1;
3054:   while(m % 2 == 0) m >>= 1, ++e;
3055:   T x = mod_pow(a, (m - 1) >> 1, p);
3056:   T y = a * (x * x % p) % p;
3057:   (x *= a) %= p;
3058:   T z = mod_pow(b, m, p);
3059:   while(y != 1) {
3060:     T j = 0, t = y;
3061:     while(t != 1) {
3062:       j += 1;
3063:       (t *= t) %= p;
3064:     }
3065:     z = mod_pow(z, T(1) << (e - j - 1), p);
3066:     (x *= z) %= p;
3067:     (z *= z) %= p;
3068:     (y *= z) %= p;
3069:     e = j;
3070:   }
3071:   return x;
3072: }
3073:
3074:
3075: ######################################################
3076: ######### arbitrary-mod-convolution.cpp ##########
3077: ######################################################
3078:
3079: template< typename T >
3080: struct ArbitraryModConvolution {
3081:   using real = FastFourierTransform::real;
3082:   using C = FastFourierTransform::C;
3083:
3084:   ArbitraryModConvolution() = default;
3085:
3086:   vector< T > multiply(const vector< T > &a, const vector< T > &b, int need = -1)
{
3087:     if(need == -1) need = a.size() + b.size() - 1;
3088:     int nbase = 0;
3089:     while((1 << nbase) < need) nbase++;
3090:     FastFourierTransform::ensure_base(nbase);
3091:     int sz = 1 << nbase;
3092:     vector< C > fa(sz);
3093:     for(int i = 0; i < a.size(); i++) {
3094:       fa[i] = C(a[i].x & ((1 << 15) - 1), a[i].x >> 15);
3095:     }
3096:     fft(fa, sz);
3097:     vector< C > fb(sz);
3098:     if(a == b) {
3099:       fb = fa;
3100:     } else {
3101:       for(int i = 0; i < b.size(); i++) {
3102:         fb[i] = C(b[i].x & ((1 << 15) - 1), b[i].x >> 15);
3103:       }
3104:       fft(fb, sz);
3105:     }
3106:     real ratio = 0.25 / sz;
3107:     C r2(0, -1), r3(ratio, 0), r4(0, -ratio), r5(0, 1);
3108:     for(int i = 0; i <= (sz >> 1); i++) {
3109:       int j = (sz - i) & (sz - 1);
3110:       C a1 = (fa[i] + fa[j].conj());
3111:       C a2 = (fa[i] - fa[j].conj()) * r2;
3112:       C b1 = (fb[i] + fb[j].conj()) * r3;
3113:       C b2 = (fb[i] - fb[j].conj()) * r4;
3114:       if(i != j) {
3115:         C c1 = (fa[j] + fa[i].conj());
```

```
3116:         C c2 = (fa[j] - fa[i].conj()) * r2;
3117:         C d1 = (fb[j] + fb[i].conj()) * r3;
3118:         C d2 = (fb[j] - fb[i].conj()) * r4;
3119:         fa[i] = c1 * d1 + c2 * d2 * r5;
3120:         fb[i] = c1 * d2 + c2 * d1;
3121:       }
3122:       fa[j] = a1 * b1 + a2 * b2 * r5;
3123:       fb[j] = a1 * b2 + a2 * b1;
3124:     }
3125:     fft(fa, sz);
3126:     fft(fb, sz);
3127:     vector< T > ret(need);
3128:     for(int i = 0; i < need; i++) {
3129:       int64_t aa = llround(fa[i].x);
3130:       int64_t bb = llround(fb[i].x);
3131:       int64_t cc = llround(fa[i].y);
3132:       aa = T(aa).x, bb = T(bb).x, cc = T(cc).x;
3133:       ret[i] = aa + (bb << 15) + (cc << 30);
3134:     }
3135:     return ret;
3136:   }
3137: };
3138:
3139:
3140: ################################################
3141:  number-theoretic-transform-friendly-mod-int.cpp #
3142: ################################################
3143:
3144: template< typename Mint >
3145: struct NumberTheoreticTransformFriendlyModInt {
3146:
3147:   vector< int > rev;
3148:   vector< Mint > rts;
3149:   int base, max_base;
3150:   Mint root;
3151:
3152:   NumberTheoreticTransformFriendlyModInt() : base(1), rev{0, 1}, rts{0, 1} {
3153:     const int mod = Mint::get_mod();
3154:     assert(mod >= 3 && mod % 2 == 1);
3155:     auto tmp = mod - 1;
3156:     max_base = 0;
3157:     while(tmp % 2 == 0) tmp >>= 1, max_base++;
3158:     root = 2;
3159:     while(root.pow((mod - 1) >> 1) == 1) root += 1;
3160:     assert(root.pow(mod - 1) == 1);
3161:     root = root.pow((mod - 1) >> max_base);
3162:   }
3163:
3164:   void ensure_base(int nbase) {
3165:     if(nbase <= base) return;
3166:     rev.resize(1 << nbase);
3167:     rts.resize(1 << nbase);
3168:     for(int i = 0; i < (1 << nbase); i++) {
3169:       rev[i] = (rev[i >> 1] >> 1) + ((i & 1) << (nbase - 1));
3170:     }
3171:     assert(nbase <= max_base);
3172:     while(base < nbase) {
3173:       Mint z = root.pow(1 << (max_base - 1 - base));
3174:       for(int i = 1 << (base - 1); i < (1 << base); i++) {
3175:         rts[i << 1] = rts[i];
3176:         rts[(i << 1) + 1] = rts[i] * z;
3177:       }
3178:       ++base;
3179:     }
3180:   }
3181:
3182:
3183:   void ntt(vector< Mint > &a) {
3184:     const int n = (int) a.size();
3185:     assert((n & (n - 1)) == 0);
```

```
3186:     int zeros = __builtin_ctz(n);
3187:     ensure_base(zeros);
3188:     int shift = base - zeros;
3189:     for(int i = 0; i < n; i++) {
3190:       if(i < (rev[i] >> shift)) {
3191:         swap(a[i], a[rev[i] >> shift]);
3192:       }
3193:     }
3194:     for(int k = 1; k < n; k <<= 1) {
3195:       for(int i = 0; i < n; i += 2 * k) {
3196:         for(int j = 0; j < k; j++) {
3197:           Mint z = a[i + j + k] * rts[j + k];
3198:           a[i + j + k] = a[i + j] - z;
3199:           a[i + j] = a[i + j] + z;
3200:         }
3201:       }
3202:     }
3203:   }
3204:
3205:
3206:   void intt(vector< Mint > &a) {
3207:     const int n = (int) a.size();
3208:     ntt(a);
3209:     reverse(a.begin() + 1, a.end());
3210:     Mint inv_sz = Mint(1) / n;
3211:     for(int i = 0; i < n; i++) a[i] *= inv_sz;
3212:   }
3213:
3214:   vector< Mint > multiply(vector< Mint > a, vector< Mint > b) {
3215:     int need = a.size() + b.size() - 1;
3216:     int nbase = 1;
3217:     while((1 << nbase) < need) nbase++;
3218:     ensure_base(nbase);
3219:     int sz = 1 << nbase;
3220:     a.resize(sz, 0);
3221:     b.resize(sz, 0);
3222:     ntt(a);
3223:     ntt(b);
3224:     Mint inv_sz = Mint(1) / sz;
3225:     for(int i = 0; i < sz; i++) {
3226:       a[i] *= b[i] * inv_sz;
3227:     }
3228:     reverse(a.begin() + 1, a.end());
3229:     ntt(a);
3230:     a.resize(need);
3231:     return a;
3232:   }
3233: };
3234:
3235:
3236: ################################################
3237: ################## euler-phi.cpp ###############
3238: ################################################
3239:
3240: int64_t euler_phi(int64_t n) {
3241:   int64_t ret = n;
3242:   for(int64_t i = 2; i * i <= n; i++) {
3243:     if(n % i == 0) {
3244:       ret -= ret / i;
3245:       while(n % i == 0) n /= i;
3246:     }
3247:   }
3248:   if(n > 1) ret -= ret / n;
3249:   return ret;
3250: }
3251:
3252:
3253: ################################################
3254: ############ lagrange-polynomial.cpp ###########
3255: ################################################
```

```
3256:
3257: template< typename T >
3258: T lagrange_polynomial(const vector< T > &y, int64_t t) {
3259:   int N = y.size() - 1;
3260:   Combination< T > comb(N);
3261:   if(t <= N) return y[t];
3262:   T ret(0);
3263:   vector< T > dp(N + 1, 1), pd(N + 1, 1);
3264:   for(int i = 0; i < N; i++) dp[i + 1] = dp[i] * (t - i);
3265:   for(int i = N; i > 0; i--) pd[i - 1] = pd[i] * (t - i);
3266:   for(int i = 0; i <= N; i++) {
3267:     T tmp = y[i] * dp[i] * pd[i] * comb.rfact(i) * comb.rfact(N - i);
3268:     if((N - i) & 1) ret -= tmp;
3269:     else ret += tmp;
3270:   }
3271:   return ret;
3272: }
3273:
3274:
3275: ######################################################
3276: ################## combination.cpp ##################
3277: ######################################################
3278:
3279: template< typename T >
3280: struct Combination {
3281:   vector< T > _fact, _rfact, _inv;
3282:
3283:   Combination(int sz) : _fact(sz + 1), _rfact(sz + 1), _inv(sz + 1) {
3284:     _fact[0] = _rfact[sz] = _inv[0] = 1;
3285:     for(int i = 1; i <= sz; i++) _fact[i] = _fact[i - 1] * i;
3286:     _rfact[sz] /= _fact[sz];
3287:     for(int i = sz - 1; i >= 0; i--) _rfact[i] = _rfact[i + 1] * (i + 1);
3288:     for(int i = 1; i <= sz; i++) _inv[i] = _rfact[i] * _fact[i - 1];
3289:   }
3290:
3291:   inline T fact(int k) const { return _fact[k]; }
3292:
3293:   inline T rfact(int k) const { return _rfact[k]; }
3294:
3295:   inline T inv(int k) const { return _inv[k]; }
3296:
3297:   T P(int n, int r) const {
3298:     if(r < 0 || n < r) return 0;
3299:     return fact(n) * rfact(n - r);
3300:   }
3301:
3302:   T C(int p, int q) const {
3303:     if(q < 0 || p < q) return 0;
3304:     return fact(p) * rfact(q) * rfact(p - q);
3305:   }
3306:
3307:   T H(int n, int r) const {
3308:     if(n < 0 || r < 0) return (0);
3309:     return r == 0 ? 1 : C(n + r - 1, r);
3310:   }
3311: };
3312:
3313:
3314: ######################################################
3315: ################## mod-log.cpp ##################
3316: ######################################################
3317:
3318: int64_t mod_log(int64_t a, int64_t b, int64_t p) {
3319:   int64_t g = 1;
3320:
3321:   for(int64_t i = p; i; i /= 2) (g *= a) %= p;
3322:   g = __gcd(g, p);
3323:
3324:   int64_t t = 1, c = 0;
3325:   for(; t % g; c++) {
```

```
 3326:     if(t == b) return c;
 3327:     (t *= a) %= p;
 3328:   }
 3329:   if(b % g) return -1;
 3330:
 3331:   t /= g;
 3332:   b /= g;
 3333:
 3334:   int64_t n = p / g, h = 0, gs = 1;
 3335:
 3336:   for(; h * h < n; h++) (gs *= a) %= n;
 3337:
 3338:   unordered_map< int64_t, int64_t > bs;
 3339:   for(int64_t s = 0, e = b; s < h; bs[e] = ++s) {
 3340:     (e *= a) %= n;
 3341:   }
 3342:
 3343:   for(int64_t s = 0, e = t; s < n;) {
 3344:     (e *= gs) %= n;
 3345:     s += h;
 3346:     if(bs.count(e)) return c + s - bs[e];
 3347:   }
 3348:   return -1;
 3349: }
 3350:
 3351:
 3352: ######################################################
 3353: ############### quotient-range.cpp ###############
 3354: ######################################################
 3355:
 3356: template< typename T >
 3357: vector< pair< pair< T, T >, T > > quotient_range(T N) {
 3358:   T M;
 3359:   vector< pair< pair< T, T >, T > > ret;
 3360:   for(M = 1; M * M <= N; M++) {
 3361:     ret.emplace_back(make_pair(M, M), N / M);
 3362:   }
 3363:   for(T i = M; i >= 1; i--) {
 3364:     T L = N / (i + 1) + 1;
 3365:     T R = N / i;
 3366:     if(L <= R && ret.back().first.second < L) ret.emplace_back(make_pair(L, R), N
/ L);
 3367:   }
 3368:   return ret;
 3369: }
 3370:
 3371:
 3372: ######################################################
 3373: ########## formal-power-series-seq.cpp ###########
 3374: ######################################################
 3375:
 3376: template< typename T >
 3377: FormalPowerSeries< T > bernoulli(int N) {
 3378:   FormalPowerSeries< T > poly(N + 1);
 3379:   poly[0] = T(1);
 3380:   for(int i = 1; i <= N; i++) {
 3381:     poly[i] = poly[i - 1] / T(i + 1);
 3382:   }
 3383:   poly = poly.inv();
 3384:   T tmp(1);
 3385:   for(int i = 1; i <= N; i++) {
 3386:     tmp *= T(i);
 3387:     poly[i] *= tmp;
 3388:   }
 3389:   return poly;
 3390: }
 3391:
 3392: template< typename T >
 3393: FormalPowerSeries< T > partition(int N) {
 3394:   FormalPowerSeries< T > poly(N + 1);
```

```
 3395:   poly[0] = 1;
 3396:   for(int k = 1; k <= N; k++) {
 3397:     if(1LL * k * (3 * k + 1) / 2 <= N) poly[k * (3 * k + 1) / 2] += (k % 2 ? -1 :
1);
 3398:     if(1LL * k * (3 * k - 1) / 2 <= N) poly[k * (3 * k - 1) / 2] += (k % 2 ? -1 :
1);
 3399:   }
 3400:   return poly.inv();
 3401: }
 3402:
 3403: template< typename T >
 3404: FormalPowerSeries< T > bell(int N) {
 3405:   FormalPowerSeries< T > poly(N + 1), ret(N + 1);
 3406:   poly[1] = 1;
 3407:   poly = poly.exp();
 3408:   poly[0] -= 1;
 3409:   poly = poly.exp();
 3410:   T mul = 1;
 3411:   for(int i = 0; i <= N; i++) {
 3412:     ret[i] = poly[i] * mul;
 3413:     mul *= i + 1;
 3414:   }
 3415:   return ret;
 3416: }
 3417:
 3418: template< typename T >
 3419: FormalPowerSeries< T > stirling_first(int N) {
 3420:   if(N == 0) return {1};
 3421:   int M = N / 2;
 3422:   FormalPowerSeries< T > A = stirling_first< T >(M), B, C(N - M + 1);
 3423:
 3424:   if(N % 2 == 0) {
 3425:     B = A;
 3426:   } else {
 3427:     B.resize(M + 2);
 3428:     B[M + 1] = 1;
 3429:     for(int i = 1; i < M + 1; i++) B[i] = A[i - 1] + A[i] * M;
 3430:   }
 3431:
 3432:   T tmp = 1;
 3433:   for(int i = 0; i <= N - M; i++) {
 3434:     C[N - M - i] = T(M).pow(i) / tmp;
 3435:     B[i] *= tmp;
 3436:     tmp *= T(i + 1);
 3437:   }
 3438:   C *= B;
 3439:   tmp = 1;
 3440:   for(int i = 0; i <= N - M; i++) {
 3441:     B[i] = C[N - M + i] / tmp;
 3442:     tmp *= T(i + 1);
 3443:   }
 3444:   return A * B;
 3445: }
 3446:
 3447: template< typename T >
 3448: FormalPowerSeries< T > stirling_second(int N) {
 3449:   FormalPowerSeries< T > A(N + 1), B(N + 1);
 3450:   modint tmp = 1;
 3451:   for(int i = 0; i <= N; i++) {
 3452:     T rev = T(1) / tmp;
 3453:     A[i] = T(i).pow(N) * rev;
 3454:     B[i] = T(1) * rev;
 3455:     if(i & 1) B[i] *= -1;
 3456:     tmp *= i + 1;
 3457:   }
 3458:   return (A * B).pre(N + 1);
 3459: }
 3460:
 3461: template< typename T >
 3462: FormalPowerSeries< T > stirling_second_kth_column(int N, int K) {
```

```
3463:   FormalPowerSeries< T > poly(N + 1), ret(N + 1);
3464:   poly[1] = 1;
3465:   poly = poly.exp();
3466:   poly[0] -= 1;
3467:   poly = poly.pow(K);
3468:   T rev = 1, mul = 1;
3469:   for(int i = 2; i <= K; i++) rev *= i;
3470:   rev = T(1) / rev;
3471:   poly *= rev;
3472:   for(int i = 0; i <= N; i++) {
3473:     ret[i] = poly[i] * mul;
3474:     mul *= i + 1;
3475:   }
3476:   return ret;
3477: }
3478:
3479: template< typename T >
3480: FormalPowerSeries< T > eulerian(int N) {
3481:   vector< T > fact(N + 2), rfact(N + 2);
3482:   fact[0] = rfact[N + 1] = 1;
3483:   for(int i = 1; i <= N + 1; i++) fact[i] = fact[i - 1] * i;
3484:   rfact[N + 1] /= fact[N + 1];
3485:   for(int i = N; i >= 0; i--) rfact[i] = rfact[i + 1] * (i + 1);
3486:
3487:   FormalPowerSeries< T > A(N + 1), B(N + 1);
3488:   for(int i = 0; i <= N; i++) {
3489:     A[i] = fact[N + 1] * rfact[i] * rfact[N + 1 - i];
3490:     if(i & 1) A[i] *= -1;
3491:     B[i] = T(i + 1).pow(N);
3492:   }
3493:   return (A * B).pre(N + 1);
3494: }
3495:
3496:
3497: ////////////////////////////////////////////////
3498: ////////// fast-prime-factorization.cpp //////////
3499: ////////////////////////////////////////////////
3500:
3501: namespace FastPrimeFactorization {
3502:
3503:   template< typename word, typename dword, typename sword >
3504:   struct UnsafeMod {
3505:     UnsafeMod() : x(0) {}
3506:
3507:     UnsafeMod(word _x) : x(init(_x)) {}
3508:
3509:     bool operator==(const UnsafeMod &rhs) const {
3510:       return x == rhs.x;
3511:     }
3512:
3513:     bool operator!=(const UnsafeMod &rhs) const {
3514:       return x != rhs.x;
3515:     }
3516:
3517:     UnsafeMod &operator+=(const UnsafeMod &rhs) {
3518:       if((x += rhs.x) >= mod) x -= mod;
3519:       return *this;
3520:     }
3521:
3522:     UnsafeMod &operator-=(const UnsafeMod &rhs) {
3523:       if(sword(x -= rhs.x) < 0) x += mod;
3524:       return *this;
3525:     }
3526:
3527:     UnsafeMod &operator*=(const UnsafeMod &rhs) {
3528:       x = reduce(dword(x) * rhs.x);
3529:       return *this;
3530:     }
3531:
3532:     UnsafeMod operator+(const UnsafeMod &rhs) const {
```

```
3533:       return UnsafeMod(*this) += rhs;
3534:     }
3535:
3536:     UnsafeMod operator-(const UnsafeMod &rhs) const {
3537:       return UnsafeMod(*this) -= rhs;
3538:     }
3539:
3540:     UnsafeMod operator*(const UnsafeMod &rhs) const {
3541:       return UnsafeMod(*this) *= rhs;
3542:     }
3543:
3544:     UnsafeMod pow(uint64_t e) const {
3545:       UnsafeMod ret(1);
3546:       for(UnsafeMod base = *this; e; e >>= 1, base *= base) {
3547:         if(e & 1) ret *= base;
3548:       }
3549:       return ret;
3550:     }
3551:
3552:     word get() const {
3553:       return reduce(x);
3554:     }
3555:
3556:     static constexpr int word_bits = sizeof(word) * 8;
3557:
3558:     static word modulus() {
3559:       return mod;
3560:     }
3561:
3562:     static word init(word w) {
3563:       return reduce(dword(w) * r2);
3564:     }
3565:
3566:     static void set_mod(word m) {
3567:       mod = m;
3568:       inv = mul_inv(mod);
3569:       r2 = -dword(mod) % mod;
3570:     }
3571:
3572:     static word reduce(dword x) {
3573:       word y = word(x >> word_bits) - word((dword(word(x) * inv) * mod) >> word_bi
ts);
3574:       return sword(y) < 0 ? y + mod : y;
3575:     }
3576:
3577:     static word mul_inv(word n, int e = 6, word x = 1) {
3578:       return !e ? x : mul_inv(n, e - 1, x * (2 - x * n));
3579:     }
3580:
3581:     static word mod, inv, r2;
3582:
3583:     word x;
3584:   };
3585:
3586:   using uint128_t = __uint128_t;
3587:
3588:   using Mod64 = UnsafeMod< uint64_t, uint128_t, int64_t >;
3589:   template<> uint64_t Mod64::mod = 0;
3590:   template<> uint64_t Mod64::inv = 0;
3591:   template<> uint64_t Mod64::r2 = 0;
3592:
3593:   using Mod32 = UnsafeMod< uint32_t, uint64_t, int32_t >;
3594:   template<> uint32_t Mod32::mod = 0;
3595:   template<> uint32_t Mod32::inv = 0;
3596:   template<> uint32_t Mod32::r2 = 0;
3597:
3598:   bool miller_rabin_primality_test_uint64(uint64_t n) {
3599:     Mod64::set_mod(n);
3600:     uint64_t d = n - 1;
3601:     while(d % 2 == 0) d /= 2;
```

```
3602:     Mod64 e{1}, rev{n - 1};
3603:     // http://miller-rabin.appspot.com/  < 2^64
3604:     for(uint64_t a : {2, 325, 9375, 28178, 450775, 9780504, 1795265022}) {
3605:       if(n <= a) break;
3606:       uint64_t t = d;
3607:       Mod64 y = Mod64(a).pow(t);
3608:       while(t != n - 1 && y != e && y != rev) {
3609:         y *= y;
3610:         t *= 2;
3611:       }
3612:       if(y != rev && t % 2 == 0) return false;
3613:     }
3614:     return true;
3615:   }
3616:
3617:   bool miller_rabin_primality_test_uint32(uint32_t n) {
3618:     Mod32::set_mod(n);
3619:     uint32_t d = n - 1;
3620:     while(d % 2 == 0) d /= 2;
3621:     Mod32 e{1}, rev{n - 1};
3622:     for(uint32_t a : {2, 7, 61}) {
3623:       if(n <= a) break;
3624:       uint32_t t = d;
3625:       Mod32 y = Mod32(a).pow(t);
3626:       while(t != n - 1 && y != e && y != rev) {
3627:         y *= y;
3628:         t *= 2;
3629:       }
3630:       if(y != rev && t % 2 == 0) return false;
3631:     }
3632:     return true;
3633:   }
3634:
3635:   bool is_prime(uint64_t n) {
3636:     if(n == 2) return true;
3637:     if(n == 1 || n % 2 == 0) return false;
3638:     if(n < uint64_t(1) << 31) return miller_rabin_primality_test_uint32(n);
3639:     return miller_rabin_primality_test_uint64(n);
3640:   }
3641:
3642:   uint64_t pollard_rho(uint64_t n) {
3643:     if(is_prime(n)) return n;
3644:     if(n % 2 == 0) return 2;
3645:     Mod64::set_mod(n);
3646:     uint64_t d;
3647:     Mod64 one{1};
3648:     for(Mod64 c{one};; c += one) {
3649:       Mod64 x{2}, y{2};
3650:       do {
3651:         x = x * x + c;
3652:         y = y * y + c;
3653:         y = y * y + c;
3654:         d = __gcd((x - y).get(), n);
3655:       } while(d == 1);
3656:       if(d < n) return d;
3657:     }
3658:     assert(0);
3659:   }
3660:
3661:   vector< uint64_t > prime_factor(uint64_t n) {
3662:     if(n <= 1) return {};
3663:     uint64_t p = pollard_rho(n);
3664:     if(p == n) return {p};
3665:     auto l = prime_factor(p);
3666:     auto r = prime_factor(n / p);
3667:     copy(begin(r), end(r), back_inserter(l));
3668:     return l;
3669:   }
3670: };
3671:
```

```
3672:
3673: ####################################################
3674: ################## is-prime.cpp ##################
3675: ####################################################
3676:
3677: bool is_prime(int64_t x) {
3678:   for(int64_t i = 2; i * i <= x; i++) {
3679:     if(x % i == 0) return false;
3680:   }
3681:   return true;
3682: }
3683:
3684:
3685: ####################################################
3686: ############### sparse-matrix.cpp ################
3687: ####################################################
3688:
3689: template< typename T >
3690: using FPSGraph = vector< vector< pair< int, T > > >;
3691:
3692: template< typename T >
3693: FormalPowerSeries< T > random_poly(int n) {
3694:   mt19937 mt(1333333);
3695:   FormalPowerSeries< T > res(n);
3696:   uniform_int_distribution< int > rand(0, T::get_mod() - 1);
3697:   for(int i = 0; i < n; i++) res[i] = rand(mt);
3698:   return res;
3699: }
3700:
3701: template< typename T >
3702: FormalPowerSeries< T > next_poly(const FormalPowerSeries< T > &dp, const FPSGraph<
T > &g) {
3703:   const int N = (int) dp.size();
3704:   FormalPowerSeries< T > nxt(N);
3705:   for(int i = 0; i < N; i++) {
3706:     for(auto &p : g[i]) nxt[p.first] += p.second * dp[i];
3707:   }
3708:   return nxt;
3709: }
3710:
3711: template< typename T >
3712: FormalPowerSeries< T > minimum_poly(const FPSGraph< T > &g) {
3713:   const int N = (int) g.size();
3714:   auto dp = random_poly< T >(N), u = random_poly< T >(N);
3715:   FormalPowerSeries< T > f(2 * N);
3716:   for(int i = 0; i < 2 * N; i++) {
3717:     for(auto &p : u.dot(dp)) f[i] += p;
3718:     dp = next_poly(dp, g);
3719:   }
3720:   return berlekamp_massey(f);
3721: }
3722:
3723: /* O(N(N+S) + N log N log Q) (O(S): time complexity of nex) */
3724: template< typename T >
3725: FormalPowerSeries< T > sparse_pow(int64_t Q, FormalPowerSeries< modint > dp, const
FPSGraph< T > &g) {
3726:   const int N = (int) dp.size();
3727:   auto A = FormalPowerSeries< T >({0, 1}).pow_mod(Q, minimum_poly(g));
3728:   FormalPowerSeries< T > res(N);
3729:   for(int i = 0; i < A.size(); i++) {
3730:     res += dp * A[i];
3731:     dp = next_poly(dp, g);
3732:   }
3733:   return res;
3734: }
3735:
3736: /* O(N(N+S)) (S: none-zero elements)*/
3737: template< typename T >
3738: T sparse_determinant(FPSGraph< T > g) {
3739:   using FPS = FormalPowerSeries< T >;
```

```
3740:   int N = (int) g.size();
3741:   auto C = random_poly< T >(N);
3742:   for(int i = 0; i < N; i++) for(auto &p : g[i]) p.second *= C[i];
3743:   auto u = minimum_poly(g);
3744:   T acdet = u[0];
3745:   if(N % 2 == 0) acdet *= -1;
3746:   T cdet = 1;
3747:   for(int i = 0; i < N; i++) cdet *= C[i];
3748:   return acdet / cdet;
3749: }
3750:
3751:
3752: ######################################################
3753: ################ convert-base.cpp ################
3754: ######################################################
3755:
3756: template< typename T >
3757: vector< T > convert_base(T x, T b) {
3758:   vector< T > ret;
3759:   T t = 1, k = abs(b);
3760:   while(x) {
3761:     ret.emplace_back((x * t) % k);
3762:     if(ret.back() < 0) ret.back() += k;
3763:     x -= ret.back() * t;
3764:     x /= k;
3765:     t *= b / k;
3766:   }
3767:   if(ret.empty()) ret.emplace_back(0);
3768:   reverse(begin(ret), end(ret));
3769:   return ret;
3770: }
3771:
3772:
3773: ######################################################
3774: ############## partition-table.cpp ###############
3775: ######################################################
3776:
3777: template< typename T >
3778: vector< vector< T > > get_partition(int n, int k) {
3779:   vector< vector< T > > dp(n + 1, vector< T >(k + 1));
3780:   dp[0][0] = 1;
3781:   for(int i = 0; i <= n; i++) {
3782:     for(int j = 1; j <= k; j++) {
3783:       if(i - j >= 0) dp[i][j] = dp[i][j - 1] + dp[i - j][j];
3784:       else dp[i][j] = dp[i][j - 1];
3785:     }
3786:   }
3787:   return dp;
3788: }
3789:
3790:
3791: ######################################################
3792: ########### fast-fourier-transform.cpp ###########
3793: ######################################################
3794:
3795: namespace FastFourierTransform {
3796:   using real = double;
3797:
3798:   struct C {
3799:     real x, y;
3800:
3801:     C() : x(0), y(0) {}
3802:
3803:     C(real x, real y) : x(x), y(y) {}
3804:
3805:     inline C operator+(const C &c) const { return C(x + c.x, y + c.y); }
3806:
3807:     inline C operator-(const C &c) const { return C(x - c.x, y - c.y); }
3808:
3809:     inline C operator*(const C &c) const { return C(x * c.x - y * c.y, x * c.y + y
```

```
* c.x); }
3810:
3811:     inline C conj() const { return C(x, -y); }
3812:   };
3813:
3814:   const real PI = acosl(-1);
3815:   int base = 1;
3816:   vector< C > rts = { {0, 0},
3817:                       {1, 0} };
3818:   vector< int > rev = {0, 1};
3819:
3820:
3821:   void ensure_base(int nbase) {
3822:     if(nbase <= base) return;
3823:     rev.resize(1 << nbase);
3824:     rts.resize(1 << nbase);
3825:     for(int i = 0; i < (1 << nbase); i++) {
3826:       rev[i] = (rev[i >> 1] >> 1) + ((i & 1) << (nbase - 1));
3827:     }
3828:     while(base < nbase) {
3829:       real angle = PI * 2.0 / (1 << (base + 1));
3830:       for(int i = 1 << (base - 1); i < (1 << base); i++) {
3831:         rts[i << 1] = rts[i];
3832:         real angle_i = angle * (2 * i + 1 - (1 << base));
3833:         rts[(i << 1) + 1] = C(cos(angle_i), sin(angle_i));
3834:       }
3835:       ++base;
3836:     }
3837:   }
3838:
3839:   void fft(vector< C > &a, int n) {
3840:     assert((n & (n - 1)) == 0);
3841:     int zeros = __builtin_ctz(n);
3842:     ensure_base(zeros);
3843:     int shift = base - zeros;
3844:     for(int i = 0; i < n; i++) {
3845:       if(i < (rev[i] >> shift)) {
3846:         swap(a[i], a[rev[i] >> shift]);
3847:       }
3848:     }
3849:     for(int k = 1; k < n; k <<= 1) {
3850:       for(int i = 0; i < n; i += 2 * k) {
3851:         for(int j = 0; j < k; j++) {
3852:           C z = a[i + j + k] * rts[j + k];
3853:           a[i + j + k] = a[i + j] - z;
3854:           a[i + j] = a[i + j] + z;
3855:         }
3856:       }
3857:     }
3858:   }
3859:
3860:   vector< int64_t > multiply(const vector< int > &a, const vector< int > &b) {
3861:     int need = (int) a.size() + (int) b.size() - 1;
3862:     int nbase = 1;
3863:     while((1 << nbase) < need) nbase++;
3864:     ensure_base(nbase);
3865:     int sz = 1 << nbase;
3866:     vector< C > fa(sz);
3867:     for(int i = 0; i < sz; i++) {
3868:       int x = (i < (int) a.size() ? a[i] : 0);
3869:       int y = (i < (int) b.size() ? b[i] : 0);
3870:       fa[i] = C(x, y);
3871:     }
3872:     fft(fa, sz);
3873:     C r(0, -0.25 / (sz >> 1)), s(0, 1), t(0.5, 0);
3874:     for(int i = 0; i <= (sz >> 1); i++) {
3875:       int j = (sz - i) & (sz - 1);
3876:       C z = (fa[j] * fa[j] - (fa[i] * fa[i]).conj()) * r;
3877:       fa[j] = (fa[i] * fa[i] - (fa[j] * fa[j]).conj()) * r;
3878:       fa[i] = z;
```

```
3879:     }
3880:     for(int i = 0; i < (sz >> 1); i++) {
3881:       C A0 = (fa[i] + fa[i + (sz >> 1)]) * t;
3882:       C A1 = (fa[i] - fa[i + (sz >> 1)]) * t * rts[(sz >> 1) + i];
3883:       fa[i] = A0 + A1 * s;
3884:     }
3885:     fft(fa, sz >> 1);
3886:     vector< int64_t > ret(need);
3887:     for(int i = 0; i < need; i++) {
3888:       ret[i] = llround(i & 1 ? fa[i >> 1].y : fa[i >> 1].x);
3889:     }
3890:     return ret;
3891:   }
3892: };
3893:
3894:
3895: ////////////////////////////////////////////////
3896: ////////////////// extgcd.cpp //////////////////
3897: ////////////////////////////////////////////////
3898:
3899: template< typename T >
3900: T extgcd(T a, T b, T &x, T &y) {
3901:   T d = a;
3902:   if(b != 0) {
3903:     d = extgcd(b, a % b, y, x);
3904:     y -= (a / b) * x;
3905:   } else {
3906:     x = 1;
3907:     y = 0;
3908:   }
3909:   return d;
3910: }
3911:
3912:
3913: ////////////////////////////////////////////////
3914: ////////// formal-power-series.cpp /////////////
3915: ////////////////////////////////////////////////
3916:
3917: template< typename T >
3918: struct FormalPowerSeries : vector< T > {
3919:   using vector< T >::vector;
3920:   using P = FormalPowerSeries;
3921:
3922:   using MULT = function< P(P, P) >;
3923:   using FFT = function< void(P &) >;
3924:
3925:   static MULT &get_mult() {
3926:     static MULT mult = nullptr;
3927:     return mult;
3928:   }
3929:
3930:   static void set_mult(MULT f) { get_mult() = f; }
3931:
3932:   static FFT &get_fft() {
3933:     static FFT fft = nullptr;
3934:     return fft;
3935:   }
3936:
3937:   static FFT &get_ifft() {
3938:     static FFT ifft = nullptr;
3939:     return ifft;
3940:   }
3941:
3942:   static void set_fft(FFT f, FFT g) {
3943:     get_fft() = f;
3944:     get_ifft() = g;
3945:   }
3946:
3947:   void shrink() {
3948:     while(this->size() && this->back() == T(0)) this->pop_back();
```

```
3949:   }
3950:
3951:   P operator+(const P &r) const { return P(*this) += r; }
3952:
3953:   P operator+(const T &v) const { return P(*this) += v; }
3954:
3955:   P operator-(const P &r) const { return P(*this) -= r; }
3956:
3957:   P operator-(const T &v) const { return P(*this) -= v; }
3958:
3959:   P operator*(const P &r) const { return P(*this) *= r; }
3960:
3961:   P operator*(const T &v) const { return P(*this) *= v; }
3962:
3963:   P operator/(const P &r) const { return P(*this) /= r; }
3964:
3965:   P operator%(const P &r) const { return P(*this) %= r; }
3966:
3967:   P &operator+=(const P &r) {
3968:     if(r.size() > this->size()) this->resize(r.size());
3969:     for(int i = 0; i < r.size(); i++) (*this)[i] += r[i];
3970:     return *this;
3971:   }
3972:
3973:   P &operator+=(const T &r) {
3974:     if(this->empty()) this->resize(1);
3975:     (*this)[0] += r;
3976:     return *this;
3977:   }
3978:
3979:   P &operator-=(const P &r) {
3980:     if(r.size() > this->size()) this->resize(r.size());
3981:     for(int i = 0; i < r.size(); i++) (*this)[i] -= r[i];
3982:     shrink();
3983:     return *this;
3984:   }
3985:
3986:   P &operator-=(const T &r) {
3987:     if(this->empty()) this->resize(1);
3988:     (*this)[0] -= r;
3989:     shrink();
3990:     return *this;
3991:   }
3992:
3993:   P &operator*=(const T &v) {
3994:     const int n = (int) this->size();
3995:     for(int k = 0; k < n; k++) (*this)[k] *= v;
3996:     return *this;
3997:   }
3998:
3999:   P &operator*=(const P &r) {
4000:     if(this->empty() || r.empty()) {
4001:       this->clear();
4002:       return *this;
4003:     }
4004:     assert(get_mult() != nullptr);
4005:     return *this = get_mult()(*this, r);
4006:   }
4007:
4008:   P &operator%=(const P &r) { return *this -= *this / r * r; }
4009:
4010:   P operator-() const {
4011:     P ret(this->size());
4012:     for(int i = 0; i < this->size(); i++) ret[i] = -(*this)[i];
4013:     return ret;
4014:   }
4015:
4016:   P &operator/=(const P &r) {
4017:     if(this->size() < r.size()) {
4018:       this->clear();
```

```
4019:       return *this;
4020:     }
4021:     int n = this->size() - r.size() + 1;
4022:     return *this = (rev().pre(n) * r.rev().inv(n)).pre(n).rev(n);
4023:   }
4024:
4025:   P dot(P r) const {
4026:     P ret(min(this->size(), r.size()));
4027:     for(int i = 0; i < ret.size(); i++) ret[i] = (*this)[i] * r[i];
4028:     return ret;
4029:   }
4030:
4031:   P pre(int sz) const { return P(begin(*this), begin(*this) + min((int) this->size
(), sz)); }
4032:
4033:   P operator>>(int sz) const {
4034:     if(this->size() <= sz) return {};
4035:     P ret(*this);
4036:     ret.erase(ret.begin(), ret.begin() + sz);
4037:     return ret;
4038:   }
4039:
4040:   P operator<<(int sz) const {
4041:     P ret(*this);
4042:     ret.insert(ret.begin(), sz, T(0));
4043:     return ret;
4044:   }
4045:
4046:   P rev(int deg = -1) const {
4047:     P ret(*this);
4048:     if(deg != -1) ret.resize(deg, T(0));
4049:     reverse(begin(ret), end(ret));
4050:     return ret;
4051:   }
4052:
4053:   P diff() const {
4054:     const int n = (int) this->size();
4055:     P ret(max(0, n - 1));
4056:     for(int i = 1; i < n; i++) ret[i - 1] = (*this)[i] * T(i);
4057:     return ret;
4058:   }
4059:
4060:   P integral() const {
4061:     const int n = (int) this->size();
4062:     P ret(n + 1);
4063:     ret[0] = T(0);
4064:     for(int i = 0; i < n; i++) ret[i + 1] = (*this)[i] / T(i + 1);
4065:     return ret;
4066:   }
4067:
4068:   // F(0) must not be 0
4069:   P inv(int deg = -1) const {
4070:     assert(((*this)[0]) != T(0));
4071:     const int n = (int) this->size();
4072:     if(deg == -1) deg = n;
4073:     if(get_fft() != nullptr) {
4074:       P ret(*this);
4075:       ret.resize(deg, T(0));
4076:       return ret.inv_fast();
4077:     }
4078:     P ret({T(1) / (*this)[0]});
4079:     for(int i = 1; i < deg; i <<= 1) {
4080:       ret = (ret + ret - ret * ret * pre(i << 1)).pre(i << 1);
4081:     }
4082:     return ret.pre(deg);
4083:   }
4084:
4085:   // F(0) must be 1
4086:   P log(int deg = -1) const {
4087:     assert((*this)[0] == 1);
```

```
4088:     const int n = (int) this->size();
4089:     if(deg == -1) deg = n;
4090:     return (this->diff() * this->inv(deg)).pre(deg - 1).integral();
4091:   }
4092:
4093:   P sqrt(int deg = -1) const {
4094:     const int n = (int) this->size();
4095:     if(deg == -1) deg = n;
4096:     if((*this)[0] == T(0)) {
4097:       for(int i = 1; i < n; i++) {
4098:         if((*this)[i] != T(0)) {
4099:           if(i & 1) return {};
4100:           if(deg - i / 2 <= 0) break;
4101:           auto ret = (*this >> i).sqrt(deg - i / 2) << (i / 2);
4102:           if(ret.size() < deg) ret.resize(deg, T(0));
4103:           return ret;
4104:         }
4105:       }
4106:       return P(deg, 0);
4107:     }
4108:
4109:     P ret({T(1)});
4110:     T inv2 = T(1) / T(2);
4111:     for(int i = 1; i < deg; i <<= 1) {
4112:       ret = (ret + pre(i << 1) * ret.inv(i << 1)) * inv2;
4113:     }
4114:     return ret.pre(deg);
4115:   }
4116:
4117:   // F(0) must be 0
4118:   P exp(int deg = -1) const {
4119:     assert((*this)[0] == T(0));
4120:     const int n = (int) this->size();
4121:     if(deg == -1) deg = n;
4122:     if(get_fft() != nullptr) {
4123:       P ret(*this);
4124:       ret.resize(deg, T(0));
4125:       return ret.exp_rec();
4126:     }
4127:     P ret({T(1)});
4128:     for(int i = 1; i < deg; i <<= 1) {
4129:       ret = (ret * (pre(i << 1) + T(1) - ret.log(i << 1))).pre(i << 1);
4130:     }
4131:     return ret.pre(deg);
4132:   }
4133:
4134:
4135:   P online_convolution_exp(const P &conv_coeff) const {
4136:     const int n = (int) conv_coeff.size();
4137:     assert((n & (n - 1)) == 0);
4138:     vector< P > conv_ntt_coeff;
4139:     for(int i = n; i >= 1; i >>= 1) {
4140:       P g(conv_coeff.pre(i));
4141:       get_fft()(g);
4142:       conv_ntt_coeff.emplace_back(g);
4143:     }
4144:     P conv_arg(n), conv_ret(n);
4145:     auto rec = [&](auto rec, int l, int r, int d) -> void {
4146:       if(r - l <= 16) {
4147:         for(int i = l; i < r; i++) {
4148:           T sum = 0;
4149:           for(int j = l; j < i; j++) sum += conv_arg[j] * conv_coeff[i - j];
4150:           conv_ret[i] += sum;
4151:           conv_arg[i] = i == 0 ? T(1) : conv_ret[i] / i;
4152:         }
4153:       } else {
4154:         int m = (l + r) / 2;
4155:         rec(rec, l, m, d + 1);
4156:         P pre(r - l);
4157:         for(int i = 0; i < m - l; i++) pre[i] = conv_arg[l + i];
```

```
4158:         get_fft()(pre);
4159:         for(int i = 0; i < r - l; i++) pre[i] *= conv_ntt_coeff[d][i];
4160:         get_ifft()(pre);
4161:         for(int i = 0; i < r - m; i++) conv_ret[m + i] += pre[m + i - l];
4162:         rec(rec, m, r, d + 1);
4163:       }
4164:     };
4165:     rec(rec, 0, n, 0);
4166:     return conv_arg;
4167:   }
4168:
4169:   P exp_rec() const {
4170:     assert((*this)[0] == T(0));
4171:     const int n = (int) this->size();
4172:     int m = 1;
4173:     while(m < n) m *= 2;
4174:     P conv_coeff(m);
4175:     for(int i = 1; i < n; i++) conv_coeff[i] = (*this)[i] * i;
4176:     return online_convolution_exp(conv_coeff).pre(n);
4177:   }
4178:
4179:
4180:   P inv_fast() const {
4181:     assert(((*this)[0]) != T(0));
4182:
4183:     const int n = (int) this->size();
4184:     P res{T(1) / (*this)[0]};
4185:
4186:     for(int d = 1; d < n; d <<= 1) {
4187:       P f(2 * d), g(2 * d);
4188:       for(int j = 0; j < min(n, 2 * d); j++) f[j] = (*this)[j];
4189:       for(int j = 0; j < d; j++) g[j] = res[j];
4190:       get_fft()(f);
4191:       get_fft()(g);
4192:       for(int j = 0; j < 2 * d; j++) f[j] *= g[j];
4193:       get_ifft()(f);
4194:       for(int j = 0; j < d; j++) {
4195:         f[j] = 0;
4196:         f[j + d] = -f[j + d];
4197:       }
4198:       get_fft()(f);
4199:       for(int j = 0; j < 2 * d; j++) f[j] *= g[j];
4200:       get_ifft()(f);
4201:       for(int j = 0; j < d; j++) f[j] = res[j];
4202:       res = f;
4203:     }
4204:     return res.pre(n);
4205:   }
4206:
4207:   P pow(int64_t k, int deg = -1) const {
4208:     const int n = (int) this->size();
4209:     if(deg == -1) deg = n;
4210:     for(int i = 0; i < n; i++) {
4211:       if((*this)[i] != T(0)) {
4212:         T rev = T(1) / (*this)[i];
4213:         P ret = (((*this * rev) >> i).log() * k).exp() * ((*this)[i].pow(k));
4214:         if(i * k > deg) return P(deg, T(0));
4215:         ret = (ret << (i * k)).pre(deg);
4216:         if(ret.size() < deg) ret.resize(deg, T(0));
4217:         return ret;
4218:       }
4219:     }
4220:     return *this;
4221:   }
4222:
4223:   T eval(T x) const {
4224:     T r = 0, w = 1;
4225:     for(auto &v : *this) {
4226:       r += w * v;
4227:       w *= x;
```

```
4228:     }
4229:     return r;
4230:   }
4231:
4232:   P pow_mod(int64_t n, P mod) const {
4233:     P modinv = mod.rev().inv();
4234:     auto get_div = [&](P base) {
4235:       if(base.size() < mod.size()) {
4236:         base.clear();
4237:         return base;
4238:       }
4239:       int n = base.size() - mod.size() + 1;
4240:       return (base.rev().pre(n) * modinv.pre(n)).pre(n).rev(n);
4241:     };
4242:     P x(*this), ret{1};
4243:     while(n > 0) {
4244:       if(n & 1) {
4245:         ret *= x;
4246:         ret -= get_div(ret) * mod;
4247:       }
4248:       x *= x;
4249:       x -= get_div(x) * mod;
4250:       n >>= 1;
4251:     }
4252:     return ret;
4253:   }
4254: };
4255:
4256:
4257: ######################################################
4258: ################### matrix.cpp ###################
4259: ######################################################
4260:
4261: template< class T >
4262: struct Matrix {
4263:   vector< vector< T > > A;
4264:
4265:   Matrix() {}
4266:
4267:   Matrix(size_t n, size_t m) : A(n, vector< T >(m, 0)) {}
4268:
4269:   Matrix(size_t n) : A(n, vector< T >(n, 0)) {};
4270:
4271:   size_t height() const {
4272:     return (A.size());
4273:   }
4274:
4275:   size_t width() const {
4276:     return (A[0].size());
4277:   }
4278:
4279:   inline const vector< T > &operator[](int k) const {
4280:     return (A.at(k));
4281:   }
4282:
4283:   inline vector< T > &operator[](int k) {
4284:     return (A.at(k));
4285:   }
4286:
4287:   static Matrix I(size_t n) {
4288:     Matrix mat(n);
4289:     for(int i = 0; i < n; i++) mat[i][i] = 1;
4290:     return (mat);
4291:   }
4292:
4293:   Matrix &operator+=(const Matrix &B) {
4294:     size_t n = height(), m = width();
4295:     assert(n == B.height() && m == B.width());
4296:     for(int i = 0; i < n; i++)
4297:       for(int j = 0; j < m; j++)
```

```
4298:         (*this)[i][j] += B[i][j];
4299:       return (*this);
4300:     }
4301:
4302:     Matrix &operator-=(const Matrix &B) {
4303:       size_t n = height(), m = width();
4304:       assert(n == B.height() && m == B.width());
4305:       for(int i = 0; i < n; i++)
4306:         for(int j = 0; j < m; j++)
4307:           (*this)[i][j] -= B[i][j];
4308:       return (*this);
4309:     }
4310:
4311:     Matrix &operator*=(const Matrix &B) {
4312:       size_t n = height(), m = B.width(), p = width();
4313:       assert(p == B.height());
4314:       vector< vector< T > > C(n, vector< T >(m, 0));
4315:       for(int i = 0; i < n; i++)
4316:         for(int j = 0; j < m; j++)
4317:           for(int k = 0; k < p; k++)
4318:             C[i][j] = (C[i][j] + (*this)[i][k] * B[k][j]);
4319:       A.swap(C);
4320:       return (*this);
4321:     }
4322:
4323:     Matrix &operator^=(long long k) {
4324:       Matrix B = Matrix::I(height());
4325:       while(k > 0) {
4326:         if(k & 1) B *= *this;
4327:         *this *= *this;
4328:         k >>= 1LL;
4329:       }
4330:       A.swap(B.A);
4331:       return (*this);
4332:     }
4333:
4334:     Matrix operator+(const Matrix &B) const {
4335:       return (Matrix(*this) += B);
4336:     }
4337:
4338:     Matrix operator-(const Matrix &B) const {
4339:       return (Matrix(*this) -= B);
4340:     }
4341:
4342:     Matrix operator*(const Matrix &B) const {
4343:       return (Matrix(*this) *= B);
4344:     }
4345:
4346:     Matrix operator^(const long long k) const {
4347:       return (Matrix(*this) ^= k);
4348:     }
4349:
4350:     friend ostream &operator<<(ostream &os, Matrix &p) {
4351:       size_t n = p.height(), m = p.width();
4352:       for(int i = 0; i < n; i++) {
4353:         os << "[";
4354:         for(int j = 0; j < m; j++) {
4355:           os << p[i][j] << (j + 1 == m ? "]\n" : ",");
4356:         }
4357:       }
4358:       return (os);
4359:     }
4360:
4361:
4362:     T determinant() {
4363:       Matrix B(*this);
4364:       assert(width() == height());
4365:       T ret = 1;
4366:       for(int i = 0; i < width(); i++) {
4367:         int idx = -1;
```

```
4368:       for(int j = i; j < width(); j++) {
4369:         if(B[j][i] != 0) idx = j;
4370:       }
4371:       if(idx == -1) return (0);
4372:       if(i != idx) {
4373:         ret *= -1;
4374:         swap(B[i], B[idx]);
4375:       }
4376:       ret *= B[i][i];
4377:       T vv = B[i][i];
4378:       for(int j = 0; j < width(); j++) {
4379:         B[i][j] /= vv;
4380:       }
4381:       for(int j = i + 1; j < width(); j++) {
4382:         T a = B[j][i];
4383:         for(int k = 0; k < width(); k++) {
4384:           B[j][k] -= B[i][k] * a;
4385:         }
4386:       }
4387:     }
4388:     return (ret);
4389:   }
4390: };
4391:
4392:
4393: ######################################################
4394: ############## berlekamp-massey.cpp ##############
4395: ######################################################
4396:
4397: template< class T >
4398: FormalPowerSeries< T > berlekamp_massey(const FormalPowerSeries< T > &s) {
4399:   const int N = (int) s.size();
4400:   FormalPowerSeries< T > b = {T(-1)}, c = {T(-1)};
4401:   T y = T(1);
4402:   for(int ed = 1; ed <= N; ed++) {
4403:     int l = int(c.size()), m = int(b.size());
4404:     T x = 0;
4405:     for(int i = 0; i < l; i++) x += c[i] * s[ed - l + i];
4406:     b.emplace_back(0);
4407:     m++;
4408:     if(x == T(0)) continue;
4409:     T freq = x / y;
4410:     if(l < m) {
4411:       auto tmp = c;
4412:       c.insert(begin(c), m - l, T(0));
4413:       for(int i = 0; i < m; i++) c[m - 1 - i] -= freq * b[m - 1 - i];
4414:       b = tmp;
4415:       y = x;
4416:     } else {
4417:       for(int i = 0; i < m; i++) c[l - 1 - i] -= freq * b[m - 1 - i];
4418:     }
4419:   }
4420:   return c;
4421: }
4422:
4423:
4424: ######################################################
4425: ############## euler-phi-table.cpp ###############
4426: ######################################################
4427:
4428: vector< int > euler_phi_table(int n) {
4429:   vector< int > euler(n + 1);
4430:   for(int i = 0; i <= n; i++) {
4431:     euler[i] = i;
4432:   }
4433:   for(int i = 2; i <= n; i++) {
4434:     if(euler[i] == i) {
4435:       for(int j = i; j <= n; j += i) {
4436:         euler[j] = euler[j] / i * (i - 1);
4437:       }
```

```
4438:     }
4439:   }
4440:   return euler;
4441: }
4442:
4443:
4444: ################################################
4445: ######### number-theoretic-transform.cpp #########
4446: ################################################
4447:
4448: template< int mod >
4449: struct NumberTheoreticTransform {
4450:
4451:   vector< int > rev, rts;
4452:   int base, max_base, root;
4453:
4454:   NumberTheoreticTransform() : base(1), rev{0, 1}, rts{0, 1} {
4455:     assert(mod >= 3 && mod % 2 == 1);
4456:     auto tmp = mod - 1;
4457:     max_base = 0;
4458:     while(tmp % 2 == 0) tmp >>= 1, max_base++;
4459:     root = 2;
4460:     while(mod_pow(root, (mod - 1) >> 1) == 1) ++root;
4461:     assert(mod_pow(root, mod - 1) == 1);
4462:     root = mod_pow(root, (mod - 1) >> max_base);
4463:   }
4464:
4465:   inline int mod_pow(int x, int n) {
4466:     int ret = 1;
4467:     while(n > 0) {
4468:       if(n & 1) ret = mul(ret, x);
4469:       x = mul(x, x);
4470:       n >>= 1;
4471:     }
4472:     return ret;
4473:   }
4474:
4475:   inline int inverse(int x) {
4476:     return mod_pow(x, mod - 2);
4477:   }
4478:
4479:   inline unsigned add(unsigned x, unsigned y) {
4480:     x += y;
4481:     if(x >= mod) x -= mod;
4482:     return x;
4483:   }
4484:
4485:   inline unsigned mul(unsigned a, unsigned b) {
4486:     return 1ull * a * b % (unsigned long long) mod;
4487:   }
4488:
4489:   void ensure_base(int nbase) {
4490:     if(nbase <= base) return;
4491:     rev.resize(1 << nbase);
4492:     rts.resize(1 << nbase);
4493:     for(int i = 0; i < (1 << nbase); i++) {
4494:       rev[i] = (rev[i >> 1] >> 1) + ((i & 1) << (nbase - 1));
4495:     }
4496:     assert(nbase <= max_base);
4497:     while(base < nbase) {
4498:       int z = mod_pow(root, 1 << (max_base - 1 - base));
4499:       for(int i = 1 << (base - 1); i < (1 << base); i++) {
4500:         rts[i << 1] = rts[i];
4501:         rts[(i << 1) + 1] = mul(rts[i], z);
4502:       }
4503:       ++base;
4504:     }
4505:   }
4506:
4507:
```

```
 4508:   void ntt(vector< int > &a) {
 4509:     const int n = (int) a.size();
 4510:     assert((n & (n - 1)) == 0);
 4511:     int zeros = __builtin_ctz(n);
 4512:     ensure_base(zeros);
 4513:     int shift = base - zeros;
 4514:     for(int i = 0; i < n; i++) {
 4515:       if(i < (rev[i] >> shift)) {
 4516:         swap(a[i], a[rev[i] >> shift]);
 4517:       }
 4518:     }
 4519:     for(int k = 1; k < n; k <<= 1) {
 4520:       for(int i = 0; i < n; i += 2 * k) {
 4521:         for(int j = 0; j < k; j++) {
 4522:           int z = mul(a[i + j + k], rts[j + k]);
 4523:           a[i + j + k] = add(a[i + j], mod - z);
 4524:           a[i + j] = add(a[i + j], z);
 4525:         }
 4526:       }
 4527:     }
 4528:   }
 4529:
 4530:
 4531:   vector< int > multiply(vector< int > a, vector< int > b) {
 4532:     int need = a.size() + b.size() - 1;
 4533:     int nbase = 1;
 4534:     while((1 << nbase) < need) nbase++;
 4535:     ensure_base(nbase);
 4536:     int sz = 1 << nbase;
 4537:     a.resize(sz, 0);
 4538:     b.resize(sz, 0);
 4539:     ntt(a);
 4540:     ntt(b);
 4541:     int inv_sz = inverse(sz);
 4542:     for(int i = 0; i < sz; i++) {
 4543:       a[i] = mul(a[i], mul(b[i], inv_sz));
 4544:     }
 4545:     reverse(a.begin() + 1, a.end());
 4546:     ntt(a);
 4547:     a.resize(need);
 4548:     return a;
 4549:   }
 4550: };
 4551:
 4552:
 4553: ////////////////////////////////////////////////////
 4554: ////////////// arbitrary-mod-int.cpp ///////////////
 4555: ////////////////////////////////////////////////////
 4556:
 4557: struct ArbitraryModInt {
 4558:
 4559:   int x;
 4560:
 4561:   ArbitraryModInt() : x(0) {}
 4562:
 4563:   ArbitraryModInt(int64_t y) : x(y >= 0 ? y % mod() : (mod() - (-y) % mod()) % mod
()) {}
 4564:
 4565:   static int &mod() {
 4566:     static int mod = 0;
 4567:     return mod;
 4568:   }
 4569:
 4570:   static int set_mod(int md) {
 4571:     mod() = md;
 4572:   }
 4573:
 4574:   ArbitraryModInt &operator+=(const ArbitraryModInt &p) {
 4575:     if((x += p.x) >= mod()) x -= mod();
 4576:     return *this;
```

```
 4577:   }
 4578:
 4579:   ArbitraryModInt &operator-=(const ArbitraryModInt &p) {
 4580:     if((x += mod() - p.x) >= mod()) x -= mod();
 4581:     return *this;
 4582:   }
 4583:
 4584:   ArbitraryModInt &operator*=(const ArbitraryModInt &p) {
 4585:     unsigned long long a = (unsigned long long) x * p.x;
 4586:     unsigned xh = (unsigned) (a >> 32), xl = (unsigned) a, d, m;
 4587:     asm("divl %4; \n\t" : "=a" (d), "=d" (m) : "d" (xh), "a" (xl), "r" (mod()));
 4588:     x = m;
 4589:     return *this;
 4590:   }
 4591:
 4592:   ArbitraryModInt &operator/=(const ArbitraryModInt &p) {
 4593:     *this *= p.inverse();
 4594:     return *this;
 4595:   }
 4596:
 4597:   ArbitraryModInt operator-() const { return ArbitraryModInt(-x); }
 4598:
 4599:   ArbitraryModInt operator+(const ArbitraryModInt &p) const { return ArbitraryModI
nt(*this) += p; }
 4600:
 4601:   ArbitraryModInt operator-(const ArbitraryModInt &p) const { return ArbitraryModI
nt(*this) -= p; }
 4602:
 4603:   ArbitraryModInt operator*(const ArbitraryModInt &p) const { return ArbitraryModI
nt(*this) *= p; }
 4604:
 4605:   ArbitraryModInt operator/(const ArbitraryModInt &p) const { return ArbitraryModI
nt(*this) /= p; }
 4606:
 4607:   bool operator==(const ArbitraryModInt &p) const { return x == p.x; }
 4608:
 4609:   bool operator!=(const ArbitraryModInt &p) const { return x != p.x; }
 4610:
 4611:   ArbitraryModInt inverse() const {
 4612:     int a = x, b = mod(), u = 1, v = 0, t;
 4613:     while(b > 0) {
 4614:       t = a / b;
 4615:       swap(a -= t * b, b);
 4616:       swap(u -= t * v, v);
 4617:     }
 4618:     return ArbitraryModInt(u);
 4619:   }
 4620:
 4621:   ArbitraryModInt pow(int64_t n) const {
 4622:     ArbitraryModInt ret(1), mul(x);
 4623:     while(n > 0) {
 4624:       if(n & 1) ret *= mul;
 4625:       mul *= mul;
 4626:       n >>= 1;
 4627:     }
 4628:     return ret;
 4629:   }
 4630:
 4631:   friend ostream &operator<<(ostream &os, const ArbitraryModInt &p) {
 4632:     return os << p.x;
 4633:   }
 4634:
 4635:   friend istream &operator>>(istream &is, ArbitraryModInt &a) {
 4636:     int64_t t;
 4637:     is >> t;
 4638:     a = ArbitraryModInt(t);
 4639:     return (is);
 4640:   }
 4641: };
 4642:
```

```
 4643:
 4644: ------------------------------------------------
 4645: |                       dp                     |
 4646: ------------------------------------------------
 4647:
 4648: ################################################
 4649: ############ knapsack-limitations.cpp ############
 4650: ################################################
 4651:
 4652: template< typename T, typename Compare = greater< T > >
 4653: vector< T > knapsack_limitations(const vector< int > &w, const vector< int > &m, c
onst vector< T > &v,
 4654:                                  const int &W, const T &NG, const Compare &comp =
Compare()) {
 4655:   const int N = (int) w.size();
 4656:   vector< T > dp(W + 1, NG), deqv(W + 1);
 4657:   dp[0] = T();
 4658:   vector< int > deq(W + 1);
 4659:   for(int i = 0; i < N; i++) {
 4660:     for(int a = 0; a < w[i]; a++) {
 4661:       int s = 0, t = 0;
 4662:       for(int j = 0; w[i] * j + a <= W; j++) {
 4663:         if(dp[w[i] * j + a] != NG) {
 4664:           auto val = dp[w[i] * j + a] - j * v[i];
 4665:           while(s < t && comp(val, deqv[t - 1])) --t;
 4666:           deq[t] = j;
 4667:           deqv[t++] = val;
 4668:         }
 4669:         if(s < t) {
 4670:           dp[j * w[i] + a] = deqv[s] + j * v[i];
 4671:           if(deq[s] == j - m[i]) ++s;
 4672:         }
 4673:       }
 4674:     }
 4675:   }
 4676:   return dp;
 4677: }
 4678:
 4679:
 4680: ################################################
 4681: ############# online-offline-dp.cpp ##############
 4682: ################################################
 4683:
 4684: template< typename T, typename Compare = less< T > >
 4685: vector< T > online_offline_dp(int W, const function< T(int, int) > &f, const Compa
re &comp = Compare()) {
 4686:   vector< T > dp(W + 1);
 4687:   vector< int > isset(W + 1);
 4688:   int y_base = -1, x_base = -1;
 4689:   function< T(int, int) > get_cost = [&](int y, int x) { // return dp[0, x+x_base)
+f[x+x_base, y+y_base)
 4690:     return dp[x + x_base] + f(x + x_base, y + y_base);
 4691:   };
 4692:   function< void(int, int, int) > induce = [&](int l, int m, int r) { // dp[l, m)
-> dp[m, r)
 4693:     x_base = l, y_base = m;
 4694:     auto ret = monotone_minima(r - m, m - l, get_cost, comp);
 4695:     for(int i = 0; i < ret.size(); i++) {
 4696:       if(!isset[m + i] || comp(ret[i].second, dp[m + i])) {
 4697:         isset[m + i] = true;
 4698:         dp[m + i] = ret[i].second;
 4699:       }
 4700:     }
 4701:   };
 4702:   function< void(int, int) > dfs = [&](int l, int r) {
 4703:     if(l + 1 == r) {
 4704:       x_base = l, y_base = l;
 4705:       T cst = l ? get_cost(0, -1) : 0;
 4706:       if(!isset[l] || comp(cst, dp[l])) {
 4707:         isset[l] = true;
```

```
4708:         dp[l] = cst;
4709:       }
4710:     } else {
4711:       int mid = (l + r) / 2;
4712:       dfs(l, mid);
4713:       induce(l, mid, r);
4714:       dfs(mid, r);
4715:     }
4716:   };
4717:   dfs(0, W + 1);
4718:   return dp;
4719: };
4720:
4721:
4722: ##################################################
4723: ############### cumulative-sum.cpp ###############
4724: ##################################################
4725:
4726: template< class T >
4727: struct CumulativeSum {
4728:   vector< T > data;
4729:
4730:   CumulativeSum(int sz) : data(sz, 0) {};
4731:
4732:   void add(int k, T x) {
4733:     data[k] += x;
4734:   }
4735:
4736:   void build() {
4737:     for(int i = 1; i < data.size(); i++) {
4738:       data[i] += data[i - 1];
4739:     }
4740:   }
4741:
4742:   T query(int k) {
4743:     if(k < 0) return (0);
4744:     return (data[min(k, (int) data.size() - 1)]);
4745:   }
4746: };
4747:
4748:
4749: ##################################################
4750: ############# cumulative-sum-2d.cpp ##############
4751: ##################################################
4752:
4753: template< class T >
4754: struct CumulativeSum2D {
4755:   vector< vector< T > > data;
4756:
4757:   CumulativeSum2D(int W, int H) : data(W + 1, vector< int >(H + 1, 0)) {}
4758:
4759:   void add(int x, int y, T z) {
4760:     ++x, ++y;
4761:     if(x >= data.size() || y >= data[0].size()) return;
4762:     data[x][y] += z;
4763:   }
4764:
4765:   void build() {
4766:     for(int i = 1; i < data.size(); i++) {
4767:       for(int j = 1; j < data[i].size(); j++) {
4768:         data[i][j] += data[i][j - 1] + data[i - 1][j] - data[i - 1][j - 1];
4769:       }
4770:     }
4771:   }
4772:
4773:   T query(int sx, int sy, int gx, int gy) {
4774:     return (data[gx][gy] - data[sx][gy] - data[gx][sy] + data[sx][sy]);
4775:   }
4776: };
4777:
```

```
4778:
4779: ######################################################
4780: ############# largest-rectangle.cpp ##############
4781: ######################################################
4782:
4783: template< typename T >
4784: int64_t largest_rectangle(vector< T > height)
4785: {
4786:   stack< int > st;
4787:   height.push_back(0);
4788:   vector< int > left(height.size());
4789:   int64_t ret = 0;
4790:   for(int i = 0; i < height.size(); i++) {
4791:     while(!st.empty() && height[st.top()] >= height[i]) {
4792:       ret = max(ret, (int64_t) (i - left[st.top()] - 1) * height[st.top()]);
4793:       st.pop();
4794:     }
4795:     left[i] = st.empty() ? -1 : st.top();
4796:     st.emplace(i);
4797:   }
4798:   return (ret);
4799: }
4800:
4801:
4802: ######################################################
4803: ################## slide-min.cpp ##################
4804: ######################################################
4805:
4806: template< typename T >
4807: vector< T > slide_min(const vector< T > &v, int k)
4808: {
4809:   deque< int > deq;
4810:   vector< T > ret;
4811:   for(int i = 0; i < v.size(); i++) {
4812:     while(!deq.empty() && v[deq.back()] >= v[i]) {
4813:       deq.pop_back();
4814:     }
4815:     deq.push_back(i);
4816:     if(i - k + 1 >= 0) {
4817:       ret.emplace_back(v[deq.front()]);
4818:       if(deq.front() == i - k + 1) deq.pop_front();
4819:     }
4820:   }
4821:   return ret;
4822: }
4823:
4824:
4825: ######################################################
4826: ####### longest-increasing-subsequence.cpp #######
4827: ######################################################
4828:
4829: template< typename T >
4830: size_t longest_increasing_subsequence(const vector< T > &a, bool strict) {
4831:   vector< T > lis;
4832:   for(auto &p : a) {
4833:     typename vector< T >::iterator it;
4834:     if(strict) it = lower_bound(begin(lis), end(lis), p);
4835:     else it = upper_bound(begin(lis), end(lis), p);
4836:     if(end(lis) == it) lis.emplace_back(p);
4837:     else *it = p;
4838:   }
4839:   return lis.size();
4840: }
4841:
4842:
4843: ######################################################
4844: ############### monotone-minima.cpp ###############
4845: ######################################################
4846:
4847: template< typename T, typename Compare = less< T > >
```

```
 4848: vector< pair< int, T > > monotone_minima(int H, int W, const function< T(int, int)
 > &f, const Compare &comp = Compare()) {
 4849:   vector< pair< int, T > > dp(H);
 4850:   function< void(int, int, int, int) > dfs = [&](int top, int bottom, int left, in
t right) {
 4851:     if(top > bottom) return;
 4852:     int line = (top + bottom) / 2;
 4853:     T ma;
 4854:     int mi = -1;
 4855:     for(int i = left; i <= right; i++) {
 4856:       T cst = f(line, i);
 4857:       if(mi == -1 || comp(cst, ma)) {
 4858:         ma = cst;
 4859:         mi = i;
 4860:       }
 4861:     }
 4862:     dp[line] = make_pair(mi, ma);
 4863:     dfs(top, line - 1, left, mi);
 4864:     dfs(line + 1, bottom, mi, right);
 4865:   };
 4866:   dfs(0, H - 1, 0, W - 1);
 4867:   return dp;
 4868: }
 4869:
 4870:
 4871:
 4872: ##################################################
 4873: ########## knapsack-limitations-2.cpp ##########
 4874: ##################################################
 4875:
 4876: template< typename T >
 4877: T knapsack_limitations(const vector< T > &w, const vector< T > &m, const vector< i
nt > &v,
 4878:                        const T &W) {
 4879:   const int N = (int) w.size();
 4880:   auto v_max = *max_element(begin(v), end(v));
 4881:   if(v_max == 0) return 0;
 4882:   vector< int > ma(N);
 4883:   vector< T > mb(N);
 4884:   for(int i = 0; i < N; i++) {
 4885:     ma[i] = min< T >(m[i], v_max - 1);
 4886:     mb[i] = m[i] - ma[i];
 4887:   }
 4888:   T sum = 0;
 4889:   for(int i = 0; i < N; i++) sum += ma[i] * v[i];
 4890:   auto dp = knapsack_limitations(v, ma, w, sum, T(-1), less<>());
 4891:   vector< int > ord(N);
 4892:   iota(begin(ord), end(ord), 0);
 4893:   sort(begin(ord), end(ord), [&](int a, int b) {
 4894:     return v[a] * w[b] > v[b] * w[a];
 4895:   });
 4896:   T ret = T();
 4897:   for(int i = 0; i < dp.size(); i++) {
 4898:     if(dp[i] > W || dp[i] == -1) continue;
 4899:     T rest = W - dp[i], cost = i;
 4900:     for(auto &p : ord) {
 4901:       auto get = min(mb[p], rest / w[p]);
 4902:       if(get == 0) break;
 4903:       cost += get * v[p];
 4904:       rest -= get * w[p];
 4905:     }
 4906:     ret = max(ret, cost);
 4907:   }
 4908:   return ret;
 4909: }
 4910:
 4911:
 4912: ##################################################
 4913: ###### divide-and-conquer-optimization.cpp #######
 4914: ##################################################
```

```
 4915:
 4916: template< typename T, typename Compare = less< T > >
 4917: vector< vector< T > > divide_and_conquer_optimization(int H, int W, T INF, const f
unction< T(int, int) > &f, const Compare &comp = Compare()) {
 4918:   vector< vector< T > > dp(H + 1, vector< T >(W + 1, INF));
 4919:   dp[0][0] = 0;
 4920:   for(int i = 1; i <= H; i++) {
 4921:     function< T(int, int) > get_cost = [&](int y, int x) {
 4922:       if(x >= y) return INF;
 4923:       return dp[i - 1][x] + f(x, y);
 4924:     };
 4925:     auto ret = monotone_minima(W + 1, W + 1, get_cost, comp);
 4926:     for(int j = 0; j <= W; j++) dp[i][j] = ret[j].second;
 4927:   }
 4928:   return dp;
 4929: }
 4930:
 4931:
 4932: ######################################################
 4933: ################## hu-tucker.cpp ##################
 4934: ######################################################
 4935:
 4936: te< typename Heap, typename T >
 4937: T hu_tucker(vector< T > vs, T INF) {
 4938:   int N = (int) vs.size();
 4939:   Heap heap;
 4940:   vector< typename Heap::Node * > hs(N - 1, heap.makeheap());
 4941:   vector< int > ls(N), rs(N);
 4942:   vector< T > cs(N - 1);
 4943:   using pi = pair< T, int >;
 4944:   priority_queue< pi, vector< pi >, greater< pi > > que;
 4945:   for(int i = 0; i + 1 < N; i++) {
 4946:     ls[i] = i - 1;
 4947:     rs[i] = i + 1;
 4948:     cs[i] = vs[i] + vs[i + 1];
 4949:     que.emplace(cs[i], i);
 4950:   }
 4951:   T ret = 0;
 4952:   for(int k = 0; k + 1 < N; k++) {
 4953:     T c;
 4954:     int i;
 4955:     do {
 4956:       tie(c, i) = que.top();
 4957:       que.pop();
 4958:     } while(rs[i] < 0 || cs[i] != c);
 4959:
 4960:     bool ml = false, mr = false;
 4961:     if(!heap.empty(hs[i]) && vs[i] + heap.top(hs[i]) == c) {
 4962:       heap.pop(hs[i]);
 4963:       ml = true;
 4964:     } else if(vs[i] + vs[rs[i]] == c) {
 4965:       ml = mr = true;
 4966:     } else {
 4967:       auto top = heap.pop(hs[i]);
 4968:       if(!heap.empty(hs[i]) && heap.top(hs[i]) + top == c) {
 4969:         heap.pop(hs[i]);
 4970:       } else {
 4971:         mr = true;
 4972:       }
 4973:     }
 4974:     ret += c;
 4975:     heap.push(hs[i], c);
 4976:     if(ml) vs[i] = INF;
 4977:     if(mr) vs[rs[i]] = INF;
 4978:
 4979:     if(ml && i > 0) {
 4980:       int j = ls[i];
 4981:       hs[j] = heap.merge(hs[j], hs[i]);
 4982:       rs[j] = rs[i];
 4983:       rs[i] = -1;
```

```
4984:       ls[rs[j]] = j;
4985:       i = j;
4986:     }
4987:
4988:     if(mr && rs[i] + 1 < N) {
4989:       int j = rs[i];
4990:       hs[i] = heap.merge(hs[i], hs[j]);
4991:       rs[i] = rs[j];
4992:       rs[j] = -1;
4993:       ls[rs[i]] = i;
4994:     }
4995:     cs[i] = vs[i] + vs[rs[i]];
4996:
4997:     if(!heap.empty(hs[i])) {
4998:       T top = heap.pop(hs[i]);
4999:       cs[i] = min(cs[i], min(vs[i], vs[rs[i]]) + top);
5000:       if(!heap.empty(hs[i])) cs[i] = min(cs[i], top + heap.top(hs[i]));
5001:       heap.push(hs[i], top);
5002:     }
5003:     que.emplace(cs[i], i);
5004:   }
5005:   return ret;
5006: }
5007:
5008:
5009: ------------------------------------------------
5010: |                     string                   |
5011: ------------------------------------------------
5012:
5013: ################################################
5014: ################## manacher.cpp ################
5015: ################################################
5016:
5017: vector< int > manacher(const string &s) {
5018:   vector< int > radius(s.size());
5019:   int i = 0, j = 0;
5020:   while(i < s.size()) {
5021:     while(i - j >= 0 && i + j < s.size() && s[i - j] == s[i + j]) {
5022:       ++j;
5023:     }
5024:     radius[i] = j;
5025:     int k = 1;
5026:     while(i - k >= 0 && i + k < s.size() && k + radius[i - k] < j) {
5027:       radius[i + k] = radius[i - k];
5028:       ++k;
5029:     }
5030:     i += k;
5031:     j -= k;
5032:   }
5033:   return radius;
5034: }
5035:
5036:
5037: ################################################
5038: ################ rolling-hash.cpp ##############
5039: ################################################
5040:
5041: template< unsigned mod >
5042: struct RollingHash {
5043:   vector< unsigned > hashed, power;
5044:
5045:   inline unsigned mul(unsigned a, unsigned b) const {
5046:     unsigned long long x = (unsigned long long) a * b;
5047:     unsigned xh = (unsigned) (x >> 32), xl = (unsigned) x, d, m;
5048:     asm("divl %4; \n\t" : "=a" (d), "=d" (m) : "d" (xh), "a" (xl), "r" (mod));
5049:     return m;
5050:   }
5051:
5052:   RollingHash(const string &s, unsigned base = 10007) {
5053:     int sz = (int) s.size();
```

```
5054:     hashed.assign(sz + 1, 0);
5055:     power.assign(sz + 1, 0);
5056:     power[0] = 1;
5057:     for(int i = 0; i < sz; i++) {
5058:       power[i + 1] = mul(power[i], base);
5059:       hashed[i + 1] = mul(hashed[i], base) + s[i];
5060:       if(hashed[i + 1] >= mod) hashed[i + 1] -= mod;
5061:     }
5062:   }
5063:
5064:   unsigned get(int l, int r) const {
5065:     unsigned ret = hashed[r] + mod - mul(hashed[l], power[r - l]);
5066:     if(ret >= mod) ret -= mod;
5067:     return ret;
5068:   }
5069:
5070:   unsigned connect(unsigned h1, int h2, int h2len) const {
5071:     unsigned ret = mul(h1, power[h2len]) + h2;
5072:     if(ret >= mod) ret -= mod;
5073:     return ret;
5074:   }
5075:
5076:   int LCP(const RollingHash< mod > &b, int l1, int r1, int l2, int r2) {
5077:     int len = min(r1 - l1, r2 - l2);
5078:     int low = -1, high = len + 1;
5079:     while(high - low > 1) {
5080:       int mid = (low + high) / 2;
5081:       if(get(l1, l1 + mid) == b.get(l2, l2 + mid)) low = mid;
5082:       else high = mid;
5083:     }
5084:     return (low);
5085:   }
5086: };
5087:
5088: using RH = RollingHash< 1000000007 >;
5089:
5090:
5091: ######################################################
5092: ################ aho-corasick.cpp ################
5093: ######################################################
5094:
5095: template< int char_size, int margin >
5096: struct AhoCorasick : Trie< char_size + 1, margin > {
5097:   using Trie< char_size + 1, margin >::Trie;
5098:
5099:   const int FAIL = char_size;
5100:   vector< int > correct;
5101:
5102:   void build(bool heavy = true) {
5103:     correct.resize(this->size());
5104:     for(int i = 0; i < this->size(); i++) {
5105:       correct[i] = (int) this->nodes[i].accept.size();
5106:     }
5107:     queue< int > que;
5108:     for(int i = 0; i <= char_size; i++) {
5109:       if(~this->nodes[0].nxt[i]) {
5110:         this->nodes[this->nodes[0].nxt[i]].nxt[FAIL] = 0;
5111:         que.emplace(this->nodes[0].nxt[i]);
5112:       } else {
5113:         this->nodes[0].nxt[i] = 0;
5114:       }
5115:     }
5116:     while(!que.empty()) {
5117:       auto &now = this->nodes[que.front()];
5118:       int fail = now.nxt[FAIL];
5119:       correct[que.front()] += correct[fail];
5120:       que.pop();
5121:       for(int i = 0; i < char_size; i++) {
5122:         if(~now.nxt[i]) {
5123:           this->nodes[now.nxt[i]].nxt[FAIL] = this->nodes[fail].nxt[i];
```

```
 5124:           if(heavy) {
 5125:             auto &u = this->nodes[now.nxt[i]].accept;
 5126:             auto &v = this->nodes[this->nodes[fail].nxt[i]].accept;
 5127:             vector< int > accept;
 5128:             set_union(begin(u), end(u), begin(v), end(v), back_inserter(accept));
 5129:             u = accept;
 5130:           }
 5131:           que.emplace(now.nxt[i]);
 5132:         } else {
 5133:           now.nxt[i] = this->nodes[fail].nxt[i];
 5134:         }
 5135:       }
 5136:     }
 5137:   }
 5138:
 5139:   map< int, int > match(const string &str, int now = 0) {
 5140:     map< int, int > result;
 5141:     for(auto &c : str) {
 5142:       now = this->nodes[now].nxt[c - margin];
 5143:       for(auto &v : this->nodes[now].accept) result[v] += 1;
 5144:     }
 5145:     return result;
 5146:   }
 5147:
 5148:   pair< int64_t, int > move(const char &c, int now = 0) {
 5149:     now = this->nodes[now].nxt[c - margin];
 5150:     return {correct[now], now};
 5151:   }
 5152:
 5153:   pair< int64_t, int > move(const string &str, int now = 0) {
 5154:     int64_t sum = 0;
 5155:     for(auto &c : str) {
 5156:       auto nxt = move(c, now);
 5157:       sum += nxt.first;
 5158:       now = nxt.second;
 5159:     }
 5160:     return {sum, now};
 5161:   }
 5162: };
 5163:
 5164:
 5165: ######################################################
 5166: ################ suffix-array.cpp ################
 5167: ######################################################
 5168:
 5169: struct SuffixArray {
 5170:   vector< int > SA;
 5171:   const string s;
 5172:
 5173:   SuffixArray(const string &str) : s(str) {
 5174:     SA.resize(s.size());
 5175:     iota(begin(SA), end(SA), 0);
 5176:     sort(begin(SA), end(SA), [&](int a, int b) {
 5177:       return s[a] == s[b] ? a > b : s[a] < s[b];
 5178:     });
 5179:     vector< int > classes(s.size()), c(s.begin(), s.end()), cnt(s.size());
 5180:     for(int len = 1; len < s.size(); len <<= 1) {
 5181:       for(int i = 0; i < s.size(); i++) {
 5182:         if(i > 0 && c[SA[i - 1]] == c[SA[i]] && SA[i - 1] + len < s.size() && c[SA
[i - 1] + len / 2] == c[SA[i] + len / 2]) {
 5183:           classes[SA[i]] = classes[SA[i - 1]];
 5184:         } else {
 5185:           classes[SA[i]] = i;
 5186:         }
 5187:       }
 5188:       iota(begin(cnt), end(cnt), 0);
 5189:       copy(begin(SA), end(SA), begin(c));
 5190:       for(int i = 0; i < s.size(); i++) {
 5191:         int s1 = c[i] - len;
 5192:         if(s1 >= 0) SA[cnt[classes[s1]]++] = s1;
```

```
5193:       }
5194:       classes.swap(c);
5195:     }
5196:   }
5197:
5198:   int operator[](int k) const {
5199:     return SA[k];
5200:   }
5201:
5202:   size_t size() const {
5203:     return s.size();
5204:   }
5205:
5206:   bool lt_substr(const string &t, int si = 0, int ti = 0) {
5207:     int sn = (int) s.size(), tn = (int) t.size();
5208:     while(si < sn && ti < tn) {
5209:       if(s[si] < t[ti]) return true;
5210:       if(s[si] > t[ti]) return false;
5211:       ++si, ++ti;
5212:     }
5213:     return si >= sn && ti < tn;
5214:   }
5215:
5216:   int lower_bound(const string &t) {
5217:     int low = -1, high = (int) SA.size();
5218:     while(high - low > 1) {
5219:       int mid = (low + high) / 2;
5220:       if(lt_substr(t, SA[mid])) low = mid;
5221:       else high = mid;
5222:     }
5223:     return high;
5224:   }
5225:
5226:   pair< int, int > lower_upper_bound(string &t) {
5227:     int idx = lower_bound(t);
5228:     int low = idx - 1, high = (int) SA.size();
5229:     t.back()++;
5230:     while(high - low > 1) {
5231:       int mid = (low + high) / 2;
5232:       if(lt_substr(t, SA[mid])) low = mid;
5233:       else high = mid;
5234:     }
5235:     t.back()--;
5236:     return {idx, high};
5237:   }
5238:
5239:   void output() {
5240:     for(int i = 0; i < size(); i++) {
5241:       cout << i << ": " << s.substr(SA[i]) << endl;
5242:     }
5243:   }
5244: };
5245:
5246:
5247: ////////////////////////////////////////////////
5248: //////// longest-common-prefix-array.cpp //////
5249: ////////////////////////////////////////////////
5250:
5251: struct LongestCommonPrefixArray {
5252:   const SuffixArray &SA;
5253:   vector< int > LCP, rank;
5254:
5255:   LongestCommonPrefixArray(const SuffixArray &SA) : SA(SA), LCP(SA.size()) {
5256:     rank.resize(SA.size());
5257:     for(int i = 0; i < SA.size(); i++) {
5258:       rank[SA[i]] = i;
5259:     }
5260:     for(int i = 0, h = 0; i < SA.size(); i++) {
5261:       if(rank[i] + 1 < SA.size()) {
5262:         for(int j = SA[rank[i] + 1]; max(i, j) + h < SA.size() && SA.s[i + h] == S
```

```
A.s[j + h]; ++h);
 5263:         LCP[rank[i] + 1] = h;
 5264:         if(h > 0) --h;
 5265:       }
 5266:     }
 5267:   }
 5268:
 5269:   int operator[](int k) const {
 5270:     return LCP[k];
 5271:   }
 5272:
 5273:   size_t size() const {
 5274:     return LCP.size();
 5275:   }
 5276:
 5277:   void output() {
 5278:     for(int i = 0; i < size(); i++) {
 5279:       cout << i << ": " << LCP[i] << " " << SA.s.substr(SA[i]) << endl;
 5280:     }
 5281:   }
 5282: };
 5283:
 5284:
 5285: ######################################################
 5286: ################ z-algorithm.cpp #################
 5287: ######################################################
 5288:
 5289: vector< int > z_algorithm(const string &s) {
 5290:   vector< int > prefix(s.size());
 5291:   for(int i = 1, j = 0; i < s.size(); i++) {
 5292:     if(i + prefix[i - j] < j + prefix[j]) {
 5293:       prefix[i] = prefix[i - j];
 5294:     } else {
 5295:       int k = max(0, j + prefix[j] - i);
 5296:       while(i + k < s.size() && s[k] == s[i + k]) ++k;
 5297:       prefix[i] = k;
 5298:       j = i;
 5299:     }
 5300:   }
 5301:   prefix[0] = (int) s.size();
 5302:   return prefix;
 5303: }
 5304:
 5305:
 5306: ------------------------------------------------
 5307: |                   geometry                   |
 5308: ------------------------------------------------
 5309:
 5310: ######################################################
 5311: ################## template.cpp ##################
 5312: ######################################################
 5313:
 5314: using Real = double;
 5315: using Point = complex< Real >;
 5316: const Real EPS = 1e-8, PI = acos(-1);
 5317:
 5318: inline bool eq(Real a, Real b) { return fabs(b - a) < EPS; }
 5319:
 5320: Point operator*(const Point &p, const Real &d) {
 5321:   return Point(real(p) * d, imag(p) * d);
 5322: }
 5323:
 5324: istream &operator>>(istream &is, Point &p) {
 5325:   Real a, b;
 5326:   is >> a >> b;
 5327:   p = Point(a, b);
 5328:   return is;
 5329: }
 5330:
 5331: ostream &operator<<(ostream &os, Point &p) {
```

```
5332:   return os << fixed << setprecision(10) << p.real() << " " << p.imag();
5333: }
5334:
5335: // rotate point p counterclockwise by theta rad
5336: Point rotate(Real theta, const Point &p) {
5337:   return Point(cos(theta) * p.real() - sin(theta) * p.imag(), sin(theta) * p.real(
) + cos(theta) * p.imag());
5338: }
5339:
5340: Real radian_to_degree(Real r) {
5341:   return (r * 180.0 / PI);
5342: }
5343:
5344: Real degree_to_radian(Real d) {
5345:   return (d * PI / 180.0);
5346: }
5347:
5348: // smaller angle of the a-b-c
5349: Real get_angle(const Point &a, const Point &b, const Point &c) {
5350:   const Point v(b - a), w(c - b);
5351:   Real alpha = atan2(v.imag(), v.real()), beta = atan2(w.imag(), w.real());
5352:   if(alpha > beta) swap(alpha, beta);
5353:   Real theta = (beta - alpha);
5354:   return min(theta, 2 * acos(-1) - theta);
5355: }
5356:
5357: namespace std {
5358:   bool operator<(const Point &a, const Point &b) {
5359:     return a.real() != b.real() ? a.real() < b.real() : a.imag() < b.imag();
5360:   }
5361: }
5362:
5363:
5364: struct Line {
5365:   Point a, b;
5366:
5367:   Line() = default;
5368:
5369:   Line(Point a, Point b) : a(a), b(b) {}
5370:
5371:   Line(Real A, Real B, Real C) // Ax + By = C
5372:   {
5373:     if(eq(A, 0)) a = Point(0, C / B), b = Point(1, C / B);
5374:     else if(eq(B, 0)) b = Point(C / A, 0), b = Point(C / A, 1);
5375:     else a = Point(0, C / B), b = Point(C / A, 0);
5376:   }
5377:
5378:   friend ostream &operator<<(ostream &os, Line &p) {
5379:     return os << p.a << " to " << p.b;
5380:   }
5381:
5382:   friend istream &operator>>(istream &is, Line &a) {
5383:     return is >> a.a >> a.b;
5384:   }
5385: };
5386:
5387: struct Segment : Line {
5388:   Segment() = default;
5389:
5390:   Segment(Point a, Point b) : Line(a, b) {}
5391: };
5392:
5393: struct Circle {
5394:   Point p;
5395:   Real r;
5396:
5397:   Circle() = default;
5398:
5399:   Circle(Point p, Real r) : p(p), r(r) {}
5400: };
```

```
 5401:
 5402: using Points = vector< Point >;
 5403: using Polygon = vector< Point >;
 5404: using Segments = vector< Segment >;
 5405: using Lines = vector< Line >;
 5406: using Circles = vector< Circle >;
 5407:
 5408: Real cross(const Point &a, const Point &b) {
 5409:   return real(a) * imag(b) - imag(a) * real(b);
 5410: }
 5411:
 5412: Real dot(const Point &a, const Point &b) {
 5413:   return real(a) * real(b) + imag(a) * imag(b);
 5414: }
 5415:
 5416: // http://judge.u-aizu.ac.jp/onlinejudge/description.jsp?id=CGL_1_C
 5417: int ccw(const Point &a, Point b, Point c) {
 5418:   b = b - a, c = c - a;
 5419:   if(cross(b, c) > EPS) return +1;  // "COUNTER_CLOCKWISE"
 5420:   if(cross(b, c) < -EPS) return -1; // "CLOCKWISE"
 5421:   if(dot(b, c) < 0) return +2;      // "ONLINE_BACK" c-a-b
 5422:   if(norm(b) < norm(c)) return -2;  // "ONLINE_FRONT" a-b-c
 5423:   return 0;                         // "ON_SEGMENT" a-c-b
 5424: }
 5425:
 5426: // http://judge.u-aizu.ac.jp/onlinejudge/description.jsp?id=CGL_2_A
 5427: bool parallel(const Line &a, const Line &b) {
 5428:   return eq(cross(a.b - a.a, b.b - b.a), 0.0);
 5429: }
 5430:
 5431: // http://judge.u-aizu.ac.jp/onlinejudge/description.jsp?id=CGL_2_A
 5432: bool orthogonal(const Line &a, const Line &b) {
 5433:   return eq(dot(a.a - a.b, b.a - b.b), 0.0);
 5434: }
 5435:
 5436: // http://judge.u-aizu.ac.jp/onlinejudge/description.jsp?id=CGL_1_A
 5437: Point projection(const Line &l, const Point &p) {
 5438:   double t = dot(p - l.a, l.a - l.b) / norm(l.a - l.b);
 5439:   return l.a + (l.a - l.b) * t;
 5440: }
 5441:
 5442: Point projection(const Segment &l, const Point &p) {
 5443:   double t = dot(p - l.a, l.a - l.b) / norm(l.a - l.b);
 5444:   return l.a + (l.a - l.b) * t;
 5445: }
 5446:
 5447: // http://judge.u-aizu.ac.jp/onlinejudge/description.jsp?id=CGL_1_B
 5448: Point reflection(const Line &l, const Point &p) {
 5449:   return p + (projection(l, p) - p) * 2.0;
 5450: }
 5451:
 5452: bool intersect(const Line &l, const Point &p) {
 5453:   return abs(ccw(l.a, l.b, p)) != 1;
 5454: }
 5455:
 5456: bool intersect(const Line &l, const Line &m) {
 5457:   return abs(cross(l.b - l.a, m.b - m.a)) > EPS || abs(cross(l.b - l.a, m.b - l.a)
) < EPS;
 5458: }
 5459:
 5460: bool intersect(const Segment &s, const Point &p) {
 5461:   return ccw(s.a, s.b, p) == 0;
 5462: }
 5463:
 5464: bool intersect(const Line &l, const Segment &s) {
 5465:   return cross(l.b - l.a, s.a - l.a) * cross(l.b - l.a, s.b - l.a) < EPS;
 5466: }
 5467:
 5468: Real distance(const Line &l, const Point &p);
 5469:
```

```
 5470: bool intersect(const Circle &c, const Line &l) {
 5471:   return distance(l, c.p) <= c.r + EPS;
 5472: }
 5473:
 5474: bool intersect(const Circle &c, const Point &p) {
 5475:   return abs(abs(p - c.p) - c.r) < EPS;
 5476: }
 5477:
 5478: // http://judge.u-aizu.ac.jp/onlinejudge/description.jsp?id=CGL_2_B
 5479: bool intersect(const Segment &s, const Segment &t) {
 5480:   return ccw(s.a, s.b, t.a) * ccw(s.a, s.b, t.b) <= 0 && ccw(t.a, t.b, s.a) * ccw(
t.a, t.b, s.b) <= 0;
 5481: }
 5482:
 5483: int intersect(const Circle &c, const Segment &l) {
 5484:   if(norm(projection(l, c.p) - c.p) - c.r * c.r > EPS) return 0;
 5485:   auto d1 = abs(c.p - l.a), d2 = abs(c.p - l.b);
 5486:   if(d1 < c.r + EPS && d2 < c.r + EPS) return 0;
 5487:   if(d1 < c.r - EPS && d2 > c.r + EPS || d1 > c.r + EPS && d2 < c.r - EPS) return
1;
 5488:   const Point h = projection(l, c.p);
 5489:   if(dot(l.a - h, l.b - h) < 0) return 2;
 5490:   return 0;
 5491: }
 5492:
 5493: // http://judge.u-aizu.ac.jp/onlinejudge/description.jsp?id=CGL_7_A&lang=jp
 5494: int intersect(Circle c1, Circle c2) {
 5495:   if(c1.r < c2.r) swap(c1, c2);
 5496:   Real d = abs(c1.p - c2.p);
 5497:   if(c1.r + c2.r < d) return 4;
 5498:   if(eq(c1.r + c2.r, d)) return 3;
 5499:   if(c1.r - c2.r < d) return 2;
 5500:   if(eq(c1.r - c2.r, d)) return 1;
 5501:   return 0;
 5502: }
 5503:
 5504: Real distance(const Point &a, const Point &b) {
 5505:   return abs(a - b);
 5506: }
 5507:
 5508: Real distance(const Line &l, const Point &p) {
 5509:   return abs(p - projection(l, p));
 5510: }
 5511:
 5512: Real distance(const Line &l, const Line &m) {
 5513:   return intersect(l, m) ? 0 : distance(l, m.a);
 5514: }
 5515:
 5516: Real distance(const Segment &s, const Point &p) {
 5517:   Point r = projection(s, p);
 5518:   if(intersect(s, r)) return abs(r - p);
 5519:   return min(abs(s.a - p), abs(s.b - p));
 5520: }
 5521:
 5522: // http://judge.u-aizu.ac.jp/onlinejudge/description.jsp?id=CGL_2_D
 5523: Real distance(const Segment &a, const Segment &b) {
 5524:   if(intersect(a, b)) return 0;
 5525:   return min({distance(a, b.a), distance(a, b.b), distance(b, a.a), distance(b, a.
b)});
 5526: }
 5527:
 5528: Real distance(const Line &l, const Segment &s) {
 5529:   if(intersect(l, s)) return 0;
 5530:   return min(distance(l, s.a), distance(l, s.b));
 5531: }
 5532:
 5533: Point crosspoint(const Line &l, const Line &m) {
 5534:   Real A = cross(l.b - l.a, m.b - m.a);
 5535:   Real B = cross(l.b - l.a, l.b - m.a);
 5536:   if(eq(abs(A), 0.0) && eq(abs(B), 0.0)) return m.a;
```

```
5537:   return m.a + (m.b - m.a) * B / A;
5538: }
5539:
5540: // http://judge.u-aizu.ac.jp/onlinejudge/description.jsp?id=CGL_2_C
5541: Point crosspoint(const Segment &l, const Segment &m) {
5542:   return crosspoint(Line(l), Line(m));
5543: }
5544:
5545: // http://judge.u-aizu.ac.jp/onlinejudge/description.jsp?id=CGL_7_D
5546: pair< Point, Point > crosspoint(const Circle &c, const Line l) {
5547:   Point pr = projection(l, c.p);
5548:   Point e = (l.b - l.a) / abs(l.b - l.a);
5549:   if(eq(distance(l, c.p), c.r)) return {pr, pr};
5550:   double base = sqrt(c.r * c.r - norm(pr - c.p));
5551:   return {pr - e * base, pr + e * base};
5552: }
5553:
5554: pair< Point, Point > crosspoint(const Circle &c, const Segment &l) {
5555:   Line aa = Line(l.a, l.b);
5556:   if(intersect(c, l) == 2) return crosspoint(c, aa);
5557:   auto ret = crosspoint(c, aa);
5558:   if(dot(l.a - ret.first, l.b - ret.first) < 0) ret.second = ret.first;
5559:   else ret.first = ret.second;
5560:   return ret;
5561: }
5562:
5563: // http://judge.u-aizu.ac.jp/onlinejudge/description.jsp?id=CGL_7_E
5564: pair< Point, Point > crosspoint(const Circle &c1, const Circle &c2) {
5565:   Real d = abs(c1.p - c2.p);
5566:   Real a = acos((c1.r * c1.r + d * d - c2.r * c2.r) / (2 * c1.r * d));
5567:   Real t = atan2(c2.p.imag() - c1.p.imag(), c2.p.real() - c1.p.real());
5568:   Point p1 = c1.p + Point(cos(t + a) * c1.r, sin(t + a) * c1.r);
5569:   Point p2 = c1.p + Point(cos(t - a) * c1.r, sin(t - a) * c1.r);
5570:   return {p1, p2};
5571: }
5572:
5573: // http://judge.u-aizu.ac.jp/onlinejudge/description.jsp?id=CGL_7_F
5574: // tangent of circle c through point p
5575: pair< Point, Point > tangent(const Circle &c1, const Point &p2) {
5576:   return crosspoint(c1, Circle(p2, sqrt(norm(c1.p - p2) - c1.r * c1.r)));
5577: }
5578:
5579: // http://judge.u-aizu.ac.jp/onlinejudge/description.jsp?id=CGL_7_G
5580: // common tangent of circles c1 and c2
5581: Lines tangent(Circle c1, Circle c2) {
5582:   Lines ret;
5583:   if(c1.r < c2.r) swap(c1, c2);
5584:   Real g = norm(c1.p - c2.p);
5585:   if(eq(g, 0)) return ret;
5586:   Point u = (c2.p - c1.p) / sqrt(g);
5587:   Point v = rotate(PI * 0.5, u);
5588:   for(int s : {-1, 1}) {
5589:     Real h = (c1.r + s * c2.r) / sqrt(g);
5590:     if(eq(1 - h * h, 0)) {
5591:       ret.emplace_back(c1.p + u * c1.r, c1.p + (u + v) * c1.r);
5592:     } else if(1 - h * h > 0) {
5593:       Point uu = u * h, vv = v * sqrt(1 - h * h);
5594:       ret.emplace_back(c1.p + (uu + vv) * c1.r, c2.p - (uu + vv) * c2.r * s);
5595:       ret.emplace_back(c1.p + (uu - vv) * c1.r, c2.p - (uu - vv) * c2.r * s);
5596:     }
5597:   }
5598:   return ret;
5599: }
5600:
5601: // http://judge.u-aizu.ac.jp/onlinejudge/description.jsp?id=CGL_3_B
5602: bool is_convex(const Polygon &p) {
5603:   int n = (int) p.size();
5604:   for(int i = 0; i < n; i++) {
5605:     if(ccw(p[(i + n - 1) % n], p[i], p[(i + 1) % n]) == -1) return false;
5606:   }
```

```
5607:   return true;
5608: }
5609:
5610: // http://judge.u-aizu.ac.jp/onlinejudge/description.jsp?id=CGL_4_A
5611: Polygon convex_hull(Polygon &p) {
5612:   int n = (int) p.size(), k = 0;
5613:   if(n <= 2) return p;
5614:   sort(p.begin(), p.end());
5615:   vector< Point > ch(2 * n);
5616:   for(int i = 0; i < n; ch[k++] = p[i++]) {
5617:     while(k >= 2 && cross(ch[k - 1] - ch[k - 2], p[i] - ch[k - 1]) < EPS) --k;
5618:   }
5619:   for(int i = n - 2, t = k + 1; i >= 0; ch[k++] = p[i--]) {
5620:     while(k >= t && cross(ch[k - 1] - ch[k - 2], p[i] - ch[k - 1]) < EPS) --k;
5621:   }
5622:   ch.resize(k - 1);
5623:   return ch;
5624: }
5625:
5626: // http://judge.u-aizu.ac.jp/onlinejudge/description.jsp?id=CGL_3_C
5627: enum {
5628:   OUT, ON, IN
5629: };
5630:
5631: int contains(const Polygon &Q, const Point &p) {
5632:   bool in = false;
5633:   for(int i = 0; i < Q.size(); i++) {
5634:     Point a = Q[i] - p, b = Q[(i + 1) % Q.size()] - p;
5635:     if(a.imag() > b.imag()) swap(a, b);
5636:     if(a.imag() <= 0 && 0 < b.imag() && cross(a, b) < 0) in = !in;
5637:     if(cross(a, b) == 0 && dot(a, b) <= 0) return ON;
5638:   }
5639:   return in ? IN : OUT;
5640: }
5641:
5642:
5643: // http://judge.u-aizu.ac.jp/onlinejudge/description.jsp?id=1033
5644: // deduplication of line segments
5645: void merge_segments(vector< Segment > &segs) {
5646:
5647:   auto merge_if_able = [](Segment &s1, const Segment &s2) {
5648:     if(abs(cross(s1.b - s1.a, s2.b - s2.a)) > EPS) return false;
5649:     if(ccw(s1.a, s2.a, s1.b) == 1 || ccw(s1.a, s2.a, s1.b) == -1) return false;
5650:     if(ccw(s1.a, s1.b, s2.a) == -2 || ccw(s2.a, s2.b, s1.a) == -2) return false;
5651:     s1 = Segment(min(s1.a, s2.a), max(s1.b, s2.b));
5652:     return true;
5653:   };
5654:
5655:   for(int i = 0; i < segs.size(); i++) {
5656:     if(segs[i].b < segs[i].a) swap(segs[i].a, segs[i].b);
5657:   }
5658:   for(int i = 0; i < segs.size(); i++) {
5659:     for(int j = i + 1; j < segs.size(); j++) {
5660:       if(merge_if_able(segs[i], segs[j])) {
5661:         segs[j--] = segs.back(), segs.pop_back();
5662:       }
5663:     }
5664:   }
5665: }
5666:
5667: // http://judge.u-aizu.ac.jp/onlinejudge/description.jsp?id=1033
5668: // construct a graph with the vertex of the intersection of any two line segments
5669: vector< vector< int > > segment_arrangement(vector< Segment > &segs, vector< Point
> &ps) {
5670:   vector< vector< int > > g;
5671:   int N = (int) segs.size();
5672:   for(int i = 0; i < N; i++) {
5673:     ps.emplace_back(segs[i].a);
5674:     ps.emplace_back(segs[i].b);
5675:     for(int j = i + 1; j < N; j++) {
```

```
 5676:       const Point p1 = segs[i].b - segs[i].a;
 5677:       const Point p2 = segs[j].b - segs[j].a;
 5678:       if(cross(p1, p2) == 0) continue;
 5679:       if(intersect(segs[i], segs[j])) {
 5680:         ps.emplace_back(crosspoint(segs[i], segs[j]));
 5681:       }
 5682:     }
 5683:   }
 5684:   sort(begin(ps), end(ps));
 5685:   ps.erase(unique(begin(ps), end(ps)), end(ps));
 5686:
 5687:   int M = (int) ps.size();
 5688:   g.resize(M);
 5689:   for(int i = 0; i < N; i++) {
 5690:     vector< int > vec;
 5691:     for(int j = 0; j < M; j++) {
 5692:       if(intersect(segs[i], ps[j])) {
 5693:         vec.emplace_back(j);
 5694:       }
 5695:     }
 5696:     for(int j = 1; j < vec.size(); j++) {
 5697:       g[vec[j - 1]].push_back(vec[j]);
 5698:       g[vec[j]].push_back(vec[j - 1]);
 5699:     }
 5700:   }
 5701:   return (g);
 5702: }
 5703:
 5704: // http://judge.u-aizu.ac.jp/onlinejudge/description.jsp?id=CGL_4_C
 5705: // cut with a straight line l and return a convex polygon on the left
 5706: Polygon convex_cut(const Polygon &U, Line l) {
 5707:   Polygon ret;
 5708:   for(int i = 0; i < U.size(); i++) {
 5709:     Point now = U[i], nxt = U[(i + 1) % U.size()];
 5710:     if(ccw(l.a, l.b, now) != -1) ret.push_back(now);
 5711:     if(ccw(l.a, l.b, now) * ccw(l.a, l.b, nxt) < 0) {
 5712:       ret.push_back(crosspoint(Line(now, nxt), l));
 5713:     }
 5714:   }
 5715:   return (ret);
 5716: }
 5717:
 5718: // http://judge.u-aizu.ac.jp/onlinejudge/description.jsp?id=CGL_3_A
 5719: Real area(const Polygon &p) {
 5720:   Real A = 0;
 5721:   for(int i = 0; i < p.size(); ++i) {
 5722:     A += cross(p[i], p[(i + 1) % p.size()]);
 5723:   }
 5724:   return A * 0.5;
 5725: }
 5726:
 5727: // http://judge.u-aizu.ac.jp/onlinejudge/description.jsp?id=CGL_7_H
 5728: Real area(const Polygon &p, const Circle &c) {
 5729:   if(p.size() < 3) return 0.0;
 5730:   function< Real(Circle, Point, Point) > cross_area = [&](const Circle &c, const P
oint &a, const Point &b) {
 5731:     Point va = c.p - a, vb = c.p - b;
 5732:     Real f = cross(va, vb), ret = 0.0;
 5733:     if(eq(f, 0.0)) return ret;
 5734:     if(max(abs(va), abs(vb)) < c.r + EPS) return f;
 5735:     if(distance(Segment(a, b), c.p) > c.r - EPS) return c.r * c.r * arg(vb * conj(
va));
 5736:     auto u = crosspoint(c, Segment(a, b));
 5737:     vector< Point > tot{a, u.first, u.second, b};
 5738:     for(int i = 0; i + 1 < tot.size(); i++) {
 5739:       ret += cross_area(c, tot[i], tot[i + 1]);
 5740:     }
 5741:     return ret;
 5742:   };
 5743:   Real A = 0;
```

```
  for(int i = 0; i < p.size(); i++) {
    A += cross_area(c, p[i], p[(i + 1) % p.size()]);
  }
  return A;
}

// http://judge.u-aizu.ac.jp/onlinejudge/description.jsp?id=CGL_4_B
Real convex_diameter(const Polygon &p) {
  int N = (int) p.size();
  int is = 0, js = 0;
  for(int i = 1; i < N; i++) {
    if(p[i].imag() > p[is].imag()) is = i;
    if(p[i].imag() < p[js].imag()) js = i;
  }
  Real maxdis = norm(p[is] - p[js]);

  int maxi, maxj, i, j;
  i = maxi = is;
  j = maxj = js;
  do {
    if(cross(p[(i + 1) % N] - p[i], p[(j + 1) % N] - p[j]) >= 0) {
      j = (j + 1) % N;
    } else {
      i = (i + 1) % N;
    }
    if(norm(p[i] - p[j]) > maxdis) {
      maxdis = norm(p[i] - p[j]);
      maxi = i;
      maxj = j;
    }
  } while(i != is || j != js);
  return sqrt(maxdis);
}

// http://judge.u-aizu.ac.jp/onlinejudge/description.jsp?id=CGL_5_A
Real closest_pair(Points ps) {
  if(ps.size() <= 1) throw (0);
  sort(begin(ps), end(ps));

  auto compare_y = [&](const Point &a, const Point &b) {
    return imag(a) < imag(b);
  };
  vector< Point > beet(ps.size());
  const Real INF = 1e18;

  function< Real(int, int) > rec = [&](int left, int right) {
    if(right - left <= 1) return INF;
    int mid = (left + right) >> 1;
    auto x = real(ps[mid]);
    auto ret = min(rec(left, mid), rec(mid, right));
    inplace_merge(begin(ps) + left, begin(ps) + mid, begin(ps) + right, compare_y)
;
    int ptr = 0;
    for(int i = left; i < right; i++) {
      if(abs(real(ps[i]) - x) >= ret) continue;
      for(int j = 0; j < ptr; j++) {
        auto luz = ps[i] - beet[ptr - j - 1];
        if(imag(luz) >= ret) break;
        ret = min(ret, abs(luz));
      }
      beet[ptr++] = ps[i];
    }
    return ret;
  };
  return rec(0, (int) ps.size());
}


------------------------------------------------
|                  structure                   |
```

```
5813: ------------------------------------------------
5814:
5815: ################################################
5816: ####### convex-hull-trick-add-monotone.cpp #######
5817: ################################################
5818:
5819: template< typename T, bool isMin >
5820: struct ConvexHullTrickAddMonotone {
5821: #define F first
5822: #define S second
5823:   using P = pair< T, T >;
5824:   deque< P > H;
5825:
5826:   ConvexHullTrickAddMonotone() = default;
5827:
5828:   bool empty() const { return H.empty(); }
5829:
5830:   void clear() { H.clear(); }
5831:
5832:   inline int sgn(T x) { return x == 0 ? 0 : (x < 0 ? -1 : 1); }
5833:
5834:   using D = long double;
5835:
5836:   inline bool check(const P &a, const P &b, const P &c) {
5837:     if(b.S == a.S || c.S == b.S)
5838:       return sgn(b.F - a.F) * sgn(c.S - b.S) >= sgn(c.F - b.F) * sgn(b.S - a.S);
5839:
5840:     //return (b.F-a.F)*(c.S-b.S) >= (b.S-a.S)*(c.F-b.F);
5841:     return
5842:         D(b.F - a.F) * sgn(c.S - b.S) / D(abs(b.S - a.S)) >=
5843:         D(c.F - b.F) * sgn(b.S - a.S) / D(abs(c.S - b.S));
5844:   }
5845:
5846:   void add(T a, T b) {
5847:     if(!isMin) a *= -1, b *= -1;
5848:     P line(a, b);
5849:     if(empty()) {
5850:       H.emplace_front(line);
5851:       return;
5852:     }
5853:     if(H.front().F <= a) {
5854:       if(H.front().F == a) {
5855:         if(H.front().S <= b) return;
5856:         H.pop_front();
5857:       }
5858:       while(H.size() >= 2 && check(line, H.front(), H[1])) H.pop_front();
5859:       H.emplace_front(line);
5860:     } else {
5861:       assert(a <= H.back().F);
5862:       if(H.back().F == a) {
5863:         if(H.back().S <= b) return;
5864:         H.pop_back();
5865:       }
5866:       while(H.size() >= 2 && check(H[H.size() - 2], H.back(), line)) H.pop_back();
5867:       H.emplace_back(line);
5868:     }
5869:   }
5870:
5871:   inline T get_y(const P &a, const T &x) {
5872:     return a.F * x + a.S;
5873:   }
5874:
5875:   T query(T x) {
5876:     assert(!empty());
5877:     int l = -1, r = H.size() - 1;
5878:     while(l + 1 < r) {
5879:       int m = (l + r) >> 1;
5880:       if(get_y(H[m], x) >= get_y(H[m + 1], x)) l = m;
5881:       else r = m;
5882:     }
```

```
 5883:     if(isMin) return get_y(H[r], x);
 5884:     return -get_y(H[r], x);
 5885:   }
 5886:
 5887:   T query_monotone_inc(T x) {
 5888:     assert(!empty());
 5889:     while(H.size() >= 2 && get_y(H.front(), x) >= get_y(H[1], x)) H.pop_front();
 5890:     if(isMin) return get_y(H.front(), x);
 5891:     return -get_y(H.front(), x);
 5892:   }
 5893:
 5894:   T query_monotone_dec(T x) {
 5895:     assert(!empty());
 5896:     while(H.size() >= 2 && get_y(H.back(), x) >= get_y(H[H.size() - 2], x)) H.pop_
back();
 5897:     if(isMin) return get_y(H.back(), x);
 5898:     return -get_y(H.back(), x);
 5899:   }
 5900:
 5901: #undef F
 5902: #undef S
 5903: };
 5904:
 5905:
 5906: ////////////////////////////////////////////////////
 5907: ////////////////// skew-heap.cpp ///////////////////
 5908: ////////////////////////////////////////////////////
 5909:
 5910: template< typename T, typename E = T >
 5911: struct SkewHeap {
 5912:   using G = function< T(T, E) >;
 5913:   using H = function< E(E, E) >;
 5914:
 5915:   struct Node {
 5916:     T key;
 5917:     E lazy;
 5918:     Node *l, *r;
 5919:   };
 5920:
 5921:   const bool rev;
 5922:   const G g;
 5923:   const H h;
 5924:
 5925:   SkewHeap(bool rev = false) : g([](const T &a, const E &b) { return a + b; }),
 5926:                                h([](const E &a, const E &b) { return a + b; }), re
v(rev) {}
 5927:
 5928:   SkewHeap(const G &g, const H &h, bool rev = false) : g(g), h(h), rev(rev) {}
 5929:
 5930:   Node *propagate(Node *t) {
 5931:     if(t->lazy != 0) {
 5932:       if(t->l) t->l->lazy = h(t->l->lazy, t->lazy);
 5933:       if(t->r) t->r->lazy = h(t->r->lazy, t->lazy);
 5934:       t->key = g(t->key, t->lazy);
 5935:       t->lazy = 0;
 5936:     }
 5937:     return t;
 5938:   }
 5939:
 5940:   Node *merge(Node *x, Node *y) {
 5941:     if(!x || !y) return x ? x : y;
 5942:     propagate(x), propagate(y);
 5943:     if((x->key > y->key) ^ rev) swap(x, y);
 5944:     x->r = merge(y, x->r);
 5945:     swap(x->l, x->r);
 5946:     return x;
 5947:   }
 5948:
 5949:   void push(Node *&root, const T &key) {
 5950:     root = merge(root, new Node({key, 0, nullptr, nullptr}));
```

```
 5951:   }
 5952:
 5953:   T top(Node *root) {
 5954:     return propagate(root)->key;
 5955:   }
 5956:
 5957:   T pop(Node *&root) {
 5958:     T top = propagate(root)->key;
 5959:     auto *temp = root;
 5960:     root = merge(root->l, root->r);
 5961:     delete temp;
 5962:     return top;
 5963:   }
 5964:
 5965:   bool empty(Node *root) const {
 5966:     return !root;
 5967:   }
 5968:
 5969:   void add(Node *root, const E &lazy) {
 5970:     if(root) {
 5971:       root->lazy = h(root->lazy, lazy);
 5972:       propagate(root);
 5973:     }
 5974:   }
 5975:
 5976:   Node *makeheap() {
 5977:     return nullptr;
 5978:   }
 5979: };
 5980:
 5981:
 5982: ######################################################
 5983: ############# segment-tree-2d-2.cpp ##############
 5984: ######################################################
 5985:
 5986: template< typename structure_t, typename get_t, typename update_t >
 5987: struct SegmentTree2D {
 5988:   using merge_f = function< get_t(get_t, get_t) >;
 5989:   using get_t = function< get_t(structure_t &, int) >;
 5990:   using range_update_f = function< get_t(structure_t &, int, int, update_t) >;
 5991:
 5992:   int sz;
 5993:   vector< structure_t > seg;
 5994:   const merge_f &f;
 5995:   const get_t &g;
 5996:   const range_update_f &h;
 5997:
 5998:   SegmentTree2D(int n, const merge_f &f, const get_t &g, const range_update_f &h)
: f(f), g(g), h(h) {
 5999:     sz = 1;
 6000:     while(sz < n) sz <<= 1;
 6001:     seg.resize(2 * sz - 1);
 6002:   }
 6003:
 6004:   void update(int a, int b, int lower, int upper, update_t x, int k, int l, int r)
 {
 6005:     if(r <= a || b <= l) {
 6006:       return;
 6007:     } else if(a <= l && r <= b) {
 6008:       g(seg[k], lower, upper, x);
 6009:     } else {
 6010:       update(a, b, lower, upper, x, 2 * k + 1, l, (l + r) >> 1);
 6011:       update(a, b, lower, upper, x, 2 * k + 2, (l + r) >> 1, r);
 6012:     }
 6013:   }
 6014:
 6015:   void update(int a, int b, int l, int r, update_t x) {
 6016:     update(a, b, l, r, x, 0, 0, sz);
 6017:   }
 6018:
```

```
  get_t get(int x, int y) {
    x += sz - 1;
    get_t ret = g(seg[x], y);
    while(x > 0) {
      x = (x - 1) >> 1;
      ret = f(ret, g(seg[x], y));
    }
  }
};


################################################
############# lazy-segment-tree.cpp ##############
################################################

template< typename Monoid, typename OperatorMonoid = Monoid >
struct LazySegmentTree {
  using F = function< Monoid(Monoid, Monoid) >;
  using G = function< Monoid(Monoid, OperatorMonoid) >;
  using H = function< OperatorMonoid(OperatorMonoid, OperatorMonoid) >;

  int sz, height;
  vector< Monoid > data;
  vector< OperatorMonoid > lazy;
  const F f;
  const G g;
  const H h;
  const Monoid M1;
  const OperatorMonoid OM0;


  LazySegmentTree(int n, const F f, const G g, const H h,
                  const Monoid &M1, const OperatorMonoid OM0)
      : f(f), g(g), h(h), M1(M1), OM0(OM0) {
    sz = 1;
    height = 0;
    while(sz < n) sz <<= 1, height++;
    data.assign(2 * sz, M1);
    lazy.assign(2 * sz, OM0);
  }

  void set(int k, const Monoid &x) {
    data[k + sz] = x;
  }

  void build() {
    for(int k = sz - 1; k > 0; k--) {
      data[k] = f(data[2 * k + 0], data[2 * k + 1]);
    }
  }

  inline void propagate(int k) {
    if(lazy[k] != OM0) {
      lazy[2 * k + 0] = h(lazy[2 * k + 0], lazy[k]);
      lazy[2 * k + 1] = h(lazy[2 * k + 1], lazy[k]);
      data[k] = reflect(k);
      lazy[k] = OM0;
    }
  }

  inline Monoid reflect(int k) {
    return lazy[k] == OM0 ? data[k] : g(data[k], lazy[k]);
  }

  inline void recalc(int k) {
    while(k >>= 1) data[k] = f(reflect(2 * k + 0), reflect(2 * k + 1));
  }

  inline void thrust(int k) {
    for(int i = height; i > 0; i--) propagate(k >> i);
```

```
 6089:   }
 6090:
 6091:   void update(int a, int b, const OperatorMonoid &x) {
 6092:     thrust(a += sz);
 6093:     thrust(b += sz - 1);
 6094:     for(int l = a, r = b + 1; l < r; l >>= 1, r >>= 1) {
 6095:       if(l & 1) lazy[l] = h(lazy[l], x), ++l;
 6096:       if(r & 1) --r, lazy[r] = h(lazy[r], x);
 6097:     }
 6098:     recalc(a);
 6099:     recalc(b);
 6100:   }
 6101:
 6102:   Monoid query(int a, int b) {
 6103:     thrust(a += sz);
 6104:     thrust(b += sz - 1);
 6105:     Monoid L = M1, R = M1;
 6106:     for(int l = a, r = b + 1; l < r; l >>= 1, r >>= 1) {
 6107:       if(l & 1) L = f(L, reflect(l++));
 6108:       if(r & 1) R = f(reflect(--r), R);
 6109:     }
 6110:     return f(L, R);
 6111:   }
 6112:
 6113:   Monoid operator[](const int &k) {
 6114:     return query(k, k + 1);
 6115:   }
 6116:
 6117:   template< typename C >
 6118:   int find_subtree(int a, const C &check, Monoid &M, bool type) {
 6119:     while(a < sz) {
 6120:       propagate(a);
 6121:       Monoid nxt = type ? f(reflect(2 * a + type), M) : f(M, reflect(2 * a + type)
);
 6122:       if(check(nxt)) a = 2 * a + type;
 6123:       else M = nxt, a = 2 * a + 1 - type;
 6124:     }
 6125:     return a - sz;
 6126:   }
 6127:
 6128:   template< typename C >
 6129:   int find_first(int a, const C &check) {
 6130:     Monoid L = M1;
 6131:     if(a <= 0) {
 6132:       if(check(f(L, reflect(1)))) return find_subtree(1, check, L, false);
 6133:       return -1;
 6134:     }
 6135:     thrust(a + sz);
 6136:     int b = sz;
 6137:     for(a += sz, b += sz; a < b; a >>= 1, b >>= 1) {
 6138:       if(a & 1) {
 6139:         Monoid nxt = f(L, reflect(a));
 6140:         if(check(nxt)) return find_subtree(a, check, L, false);
 6141:         L = nxt;
 6142:         ++a;
 6143:       }
 6144:     }
 6145:     return -1;
 6146:   }
 6147:
 6148:
 6149:   template< typename C >
 6150:   int find_last(int b, const C &check) {
 6151:     Monoid R = M1;
 6152:     if(b >= sz) {
 6153:       if(check(f(reflect(1), R))) return find_subtree(1, check, R, true);
 6154:       return -1;
 6155:     }
 6156:     thrust(b + sz - 1);
 6157:     int a = sz;
```

```
 6158:     for(b += sz; a < b; a >>= 1, b >>= 1) {
 6159:       if(b & 1) {
 6160:         Monoid nxt = f(reflect(--b), R);
 6161:         if(check(nxt)) return find_subtree(b, check, R, true);
 6162:         R = nxt;
 6163:       }
 6164:     }
 6165:     return -1;
 6166:   }
 6167: };
 6168:
 6169:
 6170: ######################################################
 6171: ###### partially-persistent-union-find.cpp #######
 6172: ######################################################
 6173:
 6174: struct PartiallyPersistentUnionFind {
 6175:   vector< int > data;
 6176:   vector< int > last;
 6177:   vector< vector< pair< int, int > > > add;
 6178:
 6179:   PartiallyPersistentUnionFind() {}
 6180:
 6181:   PartiallyPersistentUnionFind(int sz) : data(sz, -1), last(sz, 1e9), add(sz) {
 6182:     for(auto &vs : add) vs.emplace_back(-1, -1);
 6183:   }
 6184:
 6185:   bool unite(int t, int x, int y) {
 6186:     x = find(t, x);
 6187:     y = find(t, y);
 6188:     if(x == y) return false;
 6189:     if(data[x] > data[y]) swap(x, y);
 6190:     data[x] += data[y];
 6191:     add[x].emplace_back(t, data[x]);
 6192:     data[y] = x;
 6193:     last[y] = t;
 6194:     return true;
 6195:   }
 6196:
 6197:   int find(int t, int x) {
 6198:     if(t < last[x]) return x;
 6199:     return find(t, data[x]);
 6200:   }
 6201:
 6202:   int size(int t, int x) {
 6203:     x = find(t, x);
 6204:     return -prev(lower_bound(begin(add[x]), end(add[x]), make_pair(t, 0)))->second
;
 6205:   }
 6206: };
 6207:
 6208:
 6209: ######################################################
 6210: ############ binary-indexed-tree.cpp #############
 6211: ######################################################
 6212:
 6213: template< typename T >
 6214: struct BinaryIndexedTree {
 6215:   vector< T > data;
 6216:
 6217:   BinaryIndexedTree(int sz) {
 6218:     data.assign(++sz, 0);
 6219:   }
 6220:
 6221:   T sum(int k) {
 6222:     T ret = 0;
 6223:     for(++k; k > 0; k -= k & -k) ret += data[k];
 6224:     return (ret);
 6225:   }
 6226:
```

```
6227:   void add(int k, T x) {
6228:     for(++k; k < data.size(); k += k & -k) data[k] += x;
6229:   }
6230: };
6231:
6232:
6233: ######################################################
6234: ################## union-find.cpp ##################
6235: ######################################################
6236:
6237: struct UnionFind {
6238:   vector< int > data;
6239:
6240:   UnionFind(int sz) {
6241:     data.assign(sz, -1);
6242:   }
6243:
6244:   bool unite(int x, int y) {
6245:     x = find(x), y = find(y);
6246:     if(x == y) return (false);
6247:     if(data[x] > data[y]) swap(x, y);
6248:     data[x] += data[y];
6249:     data[y] = x;
6250:     return (true);
6251:   }
6252:
6253:   int find(int k) {
6254:     if(data[k] < 0) return (k);
6255:     return (data[k] = find(data[k]));
6256:   }
6257:
6258:   int size(int k) {
6259:     return (-data[find(k)]);
6260:   }
6261: };
6262:
6263:
6264: ######################################################
6265: ########### link-cut-tree-subtree.cpp ############
6266: ######################################################
6267:
6268: template< typename SUM, typename KEY >
6269: struct LinkCutTreeSubtree {
6270:
6271:   struct Node {
6272:     Node *l, *r, *p;
6273:
6274:     KEY key;
6275:     SUM sum;
6276:
6277:     bool rev;
6278:     int sz;
6279:
6280:     bool is_root() const {
6281:       return !p || (p->l != this && p->r != this);
6282:     }
6283:
6284:     Node(const KEY &key, const SUM &sum) :
6285:         key(key), sum(sum), rev(false), sz(1),
6286:         l(nullptr), r(nullptr), p(nullptr) {}
6287:   };
6288:
6289:   const SUM ident;
6290:
6291:   LinkCutTreeSubtree(const SUM &ident) : ident(ident) {}
6292:
6293:   Node *make_node(const KEY &key) {
6294:     auto ret = new Node(key, ident);
6295:     update(ret);
6296:     return ret;
```

```
6297:   }
6298:
6299:   Node *set_key(Node *t, const KEY &key) {
6300:     expose(t);
6301:     t->key = key;
6302:     update(t);
6303:     return t;
6304:   }
6305:
6306:   void toggle(Node *t) {
6307:     swap(t->l, t->r);
6308:     t->sum.toggle();
6309:     t->rev ^= true;
6310:   }
6311:
6312:   void push(Node *t) {
6313:     if(t->rev) {
6314:       if(t->l) toggle(t->l);
6315:       if(t->r) toggle(t->r);
6316:       t->rev = false;
6317:     }
6318:   }
6319:
6320:
6321:   void update(Node *t) {
6322:     t->sz = 1;
6323:     if(t->l) t->sz += t->l->sz;
6324:     if(t->r) t->sz += t->r->sz;
6325:     t->sum.merge(t->key, t->l ? t->l->sum : ident, t->r ? t->r->sum : ident);
6326:   }
6327:
6328:   void rotr(Node *t) {
6329:     auto *x = t->p, *y = x->p;
6330:     if((x->l = t->r)) t->r->p = x;
6331:     t->r = x, x->p = t;
6332:     update(x), update(t);
6333:     if((t->p = y)) {
6334:       if(y->l == x) y->l = t;
6335:       if(y->r == x) y->r = t;
6336:       update(y);
6337:     }
6338:   }
6339:
6340:   void rotl(Node *t) {
6341:     auto *x = t->p, *y = x->p;
6342:     if((x->r = t->l)) t->l->p = x;
6343:     t->l = x, x->p = t;
6344:     update(x), update(t);
6345:     if((t->p = y)) {
6346:       if(y->l == x) y->l = t;
6347:       if(y->r == x) y->r = t;
6348:       update(y);
6349:     }
6350:   }
6351:
6352:
6353:   void splay(Node *t) {
6354:     push(t);
6355:     while(!t->is_root()) {
6356:       auto *q = t->p;
6357:       if(q->is_root()) {
6358:         push(q), push(t);
6359:         if(q->l == t) rotr(t);
6360:         else rotl(t);
6361:       } else {
6362:         auto *r = q->p;
6363:         push(r), push(q), push(t);
6364:         if(r->l == q) {
6365:           if(q->l == t) rotr(q), rotr(t);
6366:           else rotl(t), rotr(t);
```

```
6367:         } else {
6368:           if(q->r == t) rotl(q), rotl(t);
6369:           else rotr(t), rotl(t);
6370:         }
6371:       }
6372:     }
6373:   }
6374:
6375:
6376:   Node *expose(Node *t) {
6377:     Node *rp = nullptr;
6378:     for(auto *cur = t; cur; cur = cur->p) {
6379:       splay(cur);
6380:       if(cur->r) cur->sum.add(cur->r->sum);
6381:       cur->r = rp;
6382:       if(cur->r) cur->sum.erase(cur->r->sum);
6383:       update(cur);
6384:       rp = cur;
6385:     }
6386:     splay(t);
6387:     return rp;
6388:   }
6389:
6390:   void link(Node *child, Node *parent) {
6391:     expose(child);
6392:     expose(parent);
6393:     child->p = parent;
6394:     parent->r = child;
6395:     update(parent);
6396:   }
6397:
6398:   void cut(Node *child) {
6399:     expose(child);
6400:     auto *parent = child->l;
6401:     child->l = nullptr;
6402:     parent->p = nullptr;
6403:     update(child);
6404:   }
6405:
6406:   void evert(Node *t) {
6407:     expose(t);
6408:     toggle(t);
6409:     push(t);
6410:   }
6411:
6412:   Node *lca(Node *u, Node *v) {
6413:     if(get_root(u) != get_root(v)) return nullptr;
6414:     expose(u);
6415:     return expose(v);
6416:   }
6417:
6418:
6419:   Node *get_kth(Node *x, int k) {
6420:     expose(x);
6421:     while(x) {
6422:       push(x);
6423:       if(x->r && x->r->sz > k) {
6424:         x = x->r;
6425:       } else {
6426:         if(x->r) k -= x->r->sz;
6427:         if(k == 0) return x;
6428:         k -= 1;
6429:         x = x->l;
6430:       }
6431:     }
6432:     return nullptr;
6433:   }
6434:
6435:   Node *get_root(Node *x) {
6436:     expose(x);
```

```
 6437:     while(x->l) {
 6438:       push(x);
 6439:       x = x->l;
 6440:     }
 6441:     return x;
 6442:   }
 6443:
 6444:   SUM &query(Node *t) {
 6445:     expose(t);
 6446:     return t->sum;
 6447:   }
 6448: };
 6449:
 6450:
 6451: ################################################
 6452: ##### segment-tree-fractional-cascading.cpp ######
 6453: ################################################
 6454:
 6455: struct SegmentTreeFractionalCascading {
 6456:   vector< vector< int > > seg;
 6457:   vector< vector< int > > LL, RR;
 6458:   int sz;
 6459:
 6460:   SegmentTreeFractionalCascading(vector< int > &array) {
 6461:     sz = 1;
 6462:     while(sz < array.size()) sz <<= 1;
 6463:     seg.resize(2 * sz - 1);
 6464:     LL.resize(2 * sz - 1);
 6465:     RR.resize(2 * sz - 1);
 6466:     for(int k = 0; k < array.size(); k++) {
 6467:       seg[k + sz - 1].emplace_back(array[k]);
 6468:     }
 6469:     for(int k = sz - 2; k >= 0; k--) {
 6470:       seg[k].resize(seg[2 * k + 1].size() + seg[2 * k + 2].size());
 6471:       LL[k].resize(seg[k].size() + 1);
 6472:       RR[k].resize(seg[k].size() + 1);
 6473:       merge(begin(seg[2 * k + 1]), end(seg[2 * k + 1]), begin(seg[2 * k + 2]), end
(seg[2 * k + 2]), begin(seg[k]));
 6474:       int tail1 = 0, tail2 = 0;
 6475:       for(int i = 0; i < seg[k].size(); i++) {
 6476:         while(tail1 < seg[2 * k + 1].size() && seg[2 * k + 1][tail1] < seg[k][i])
++tail1;
 6477:         while(tail2 < seg[2 * k + 2].size() && seg[2 * k + 2][tail2] < seg[k][i])
++tail2;
 6478:         LL[k][i] = tail1, RR[k][i] = tail2;
 6479:       }
 6480:       LL[k][seg[k].size()] = (int) seg[2 * k + 1].size();
 6481:       RR[k][seg[k].size()] = (int) seg[2 * k + 2].size();
 6482:     }
 6483:   }
 6484:
 6485:   int query(int a, int b, int lower, int upper, int k, int l, int r) {
 6486:     if(a >= r || b <= l) {
 6487:       return (0);
 6488:     } else if(a <= l && r <= b) {
 6489:       return (upper - lower);
 6490:     } else {
 6491:       return (query(a, b, LL[k][lower], LL[k][upper], 2 * k + 1, l, (l + r) >> 1)
+ query(a, b, RR[k][lower], RR[k][upper], 2 * k + 2, (l + r) >> 1, r));
 6492:     }
 6493:   }
 6494:
 6495:   int query(int a, int b, int l, int r) {
 6496:     l = lower_bound(begin(seg[0]), end(seg[0]), l) - begin(seg[0]);
 6497:     r = lower_bound(begin(seg[0]), end(seg[0]), r) - begin(seg[0]);
 6498:     return (query(a, b, l, r, 0, 0, sz));
 6499:   }
 6500: };
 6501:
 6502:
```

```
6503: ####################################################
6504: ############# dual-segment-tree.cpp ##############
6505: ####################################################
6506:
6507: template< typename OperatorMonoid >
6508: struct DualSegmentTree {
6509:   using H = function< OperatorMonoid(OperatorMonoid, OperatorMonoid) >;
6510:
6511:   int sz, height;
6512:   vector< OperatorMonoid > lazy;
6513:   const H h;
6514:   const OperatorMonoid OM0;
6515:
6516:
6517:   DualSegmentTree(int n, const H h, const OperatorMonoid OM0)
6518:       : h(h), OM0(OM0) {
6519:     sz = 1;
6520:     height = 0;
6521:     while(sz < n) sz <<= 1, height++;
6522:     lazy.assign(2 * sz, OM0);
6523:   }
6524:
6525:   inline void propagate(int k) {
6526:     if(lazy[k] != OM0) {
6527:       lazy[2 * k + 0] = h(lazy[2 * k + 0], lazy[k]);
6528:       lazy[2 * k + 1] = h(lazy[2 * k + 1], lazy[k]);
6529:       lazy[k] = OM0;
6530:     }
6531:   }
6532:
6533:   inline void thrust(int k) {
6534:     for(int i = height; i > 0; i--) propagate(k >> i);
6535:   }
6536:
6537:   void update(int a, int b, const OperatorMonoid &x) {
6538:     thrust(a += sz);
6539:     thrust(b += sz - 1);
6540:     for(int l = a, r = b + 1; l < r; l >>= 1, r >>= 1) {
6541:       if(l & 1) lazy[l] = h(lazy[l], x), ++l;
6542:       if(r & 1) --r, lazy[r] = h(lazy[r], x);
6543:     }
6544:   }
6545:
6546:   OperatorMonoid operator[](int k) {
6547:     thrust(k += sz);
6548:     return lazy[k];
6549:   }
6550: };
6551:
6552:
6553: ####################################################
6554: ############ weighted-union-find.cpp ##############
6555: ####################################################
6556:
6557: template< typename T >
6558: struct WeightedUnionFind {
6559:   vector< int > data;
6560:   vector< T > ws;
6561:
6562:   WeightedUnionFind() {}
6563:
6564:   WeightedUnionFind(int sz) : data(sz, -1), ws(sz) {}
6565:
6566:   int find(int k) {
6567:     if(data[k] < 0) return k;
6568:     auto par = find(data[k]);
6569:     ws[k] += ws[data[k]];
6570:     return data[k] = par;
6571:   }
6572:
```

```
  6573:   T weight(int t) {
  6574:     find(t);
  6575:     return ws[t];
  6576:   }
  6577:
  6578:   bool unite(int x, int y, T w) {
  6579:     w += weight(x);
  6580:     w -= weight(y);
  6581:     x = find(x), y = find(y);
  6582:     if(x == y) return false;
  6583:     if(data[x] > data[y]) {
  6584:       swap(x, y);
  6585:       w *= -1;
  6586:     }
  6587:     data[x] += data[y];
  6588:     data[y] = x;
  6589:     ws[y] = w;
  6590:     return true;
  6591:   }
  6592:
  6593:   T diff(int x, int y) {
  6594:     return weight(y) - weight(x);
  6595:   }
  6596: };
  6597:
  6598:
  6599: ######################################################
  6600: ################ sparse-table.cpp ################
  6601: ######################################################
  6602:
  6603: template< typename T >
  6604: struct SparseTable {
  6605:   vector< vector< T > > st;
  6606:   vector< int > lookup;
  6607:
  6608:   SparseTable(const vector< T > &v) {
  6609:     int b = 0;
  6610:     while((1 << b) <= v.size()) ++b;
  6611:     st.assign(b, vector< T >(1 << b));
  6612:     for(int i = 0; i < v.size(); i++) {
  6613:       st[0][i] = v[i];
  6614:     }
  6615:     for(int i = 1; i < b; i++) {
  6616:       for(int j = 0; j + (1 << i) <= (1 << b); j++) {
  6617:         st[i][j] = min(st[i - 1][j], st[i - 1][j + (1 << (i - 1))]);
  6618:       }
  6619:     }
  6620:     lookup.resize(v.size() + 1);
  6621:     for(int i = 2; i < lookup.size(); i++) {
  6622:       lookup[i] = lookup[i >> 1] + 1;
  6623:     }
  6624:   }
  6625:
  6626:   inline T rmq(int l, int r) {
  6627:     int b = lookup[r - l];
  6628:     return min(st[b][l], st[b][r - (1 << b)]);
  6629:   }
  6630: };
  6631:
  6632:
  6633: ######################################################
  6634: ######### persistent-red-black-tree.cpp ##########
  6635: ######################################################
  6636:
  6637: template< class D, class L, D (*f)(D, D), D (*g)(D, L), L (*h)(L, L), L (*p)(L, in
t) >
  6638: struct PersistentRedBlackTree : RedBlackTree< D, L, f, g, h, p > {
  6639:   using RBT = RedBlackTree< D, L, f, g, h, p >;
  6640:   using Node = typename RBT::Node;
  6641:
```

```
6642:   PersistentRedBlackTree(int sz, const D &M1, const L &OM0) :
6643:       RBT(sz, M1, OM0) {}
6644:
6645:   Node *clone(Node *t) override { return &(*RBT::pool.alloc() = *t); }
6646:
6647:   Node *rebuild(Node *r) {
6648:     auto ret = RBT::dump(r);
6649:     RBT::pool.clear();
6650:     return RBT::build(ret);
6651:   }
6652: };
6653:
6654:
6655: ##################################################
6656: ############## persistent-array.cpp ##############
6657: ##################################################
6658:
6659: template< typename T, int LOG >
6660: struct PersistentArray {
6661:   struct Node {
6662:     T data;
6663:     Node *child[1 << LOG] = {};
6664:
6665:     Node() {}
6666:
6667:     Node(const T &data) : data(data) {}
6668:   };
6669:
6670:   Node *root;
6671:
6672:   PersistentArray() : root(nullptr) {}
6673:
6674:   T get(Node *t, int k) {
6675:     if(k == 0) return t->data;
6676:     return get(t->child[k & ((1 << LOG) - 1)], k >> LOG);
6677:   }
6678:
6679:   T get(const int &k) {
6680:     return get(root, k);
6681:   }
6682:
6683:   pair< Node *, T * > mutable_get(Node *t, int k) {
6684:     t = t ? new Node(*t) : new Node();
6685:     if(k == 0) return {t, &t->data};
6686:     auto p = mutable_get(t->child[k & ((1 << LOG) - 1)], k >> LOG);
6687:     t->child[k & ((1 << LOG) - 1)] = p.first;
6688:     return {t, p.second};
6689:   }
6690:
6691:   T *mutable_get(const int &k) {
6692:     auto ret = mutable_get(root, k);
6693:     root = ret.first;
6694:     return ret.second;
6695:   }
6696:
6697:   Node *build(Node *t, const T &data, int k) {
6698:     if(!t) t = new Node();
6699:     if(k == 0) {
6700:       t->data = data;
6701:       return t;
6702:     }
6703:     auto p = build(t->child[k & ((1 << LOG) - 1)], data, k >> LOG);
6704:     t->child[k & ((1 << LOG) - 1)] = p;
6705:     return t;
6706:   }
6707:
6708:   void build(const vector< T > &v) {
6709:     root = nullptr;
6710:     for(int i = 0; i < v.size(); i++) {
6711:       root = build(root, v[i], i);
```

```
    }
  }
};


################################################
##### randomized-binary-search-tree-set.cpp ######
################################################

template< class T >
struct OrderedMultiSet : RandomizedBinarySearchTree< T >
{
  using RBST = RandomizedBinarySearchTree< T >;
  using Node = typename RBST::Node;

  OrderedMultiSet(int sz) : RBST(sz, [&](T x, T y) { return x; }, T()) {}

  T kth_element(Node *t, int k)
  {
    if(k < RBST::count(t->l)) return kth_element(t->l, k);
    if(k == RBST::count(t->l)) return t->key;
    return kth_element(t->r, k - RBST::count(t->l) - 1);
  }

  virtual void insert_key(Node *&t, const T &x)
  {
    RBST::insert(t, lower_bound(t, x), x);
  }

  void erase_key(Node *&t, const T &x)
  {
    if(!count(t, x)) return;
    RBST::erase(t, lower_bound(t, x));
  }

  int count(Node *t, const T &x)
  {
    return upper_bound(t, x) - lower_bound(t, x);
  }

  int lower_bound(Node *t, const T &x)
  {
    if(!t) return 0;
    if(x <= t->key) return lower_bound(t->l, x);
    return lower_bound(t->r, x) + RBST::count(t->l) + 1;
  }

  int upper_bound(Node *t, const T &x)
  {
    if(!t) return 0;
    if(x < t->key) return upper_bound(t->l, x);
    return upper_bound(t->r, x) + RBST::count(t->l) + 1;
  }
};
template< class T >
struct OrderedSet : OrderedMultiSet< T >
{
  using SET = OrderedMultiSet< T >;
  using RBST = typename SET::RBST;
  using Node = typename RBST::Node;

  OrderedSet(int sz) : OrderedMultiSet< T >(sz) {}

  void insert_key(Node *&t, const T &x) override
  {
    if(SET::count(t, x)) return;
    RBST::insert(t, SET::lower_bound(t, x), x);
  }
};
```

```
6782:
6783:
6784: ####################################################
6785: ############# segment-tree-beats.cpp #############
6786: ####################################################
6787:
6788: template< typename Monoid, typename OperatorMonoid = Monoid >
6789: struct SegmentTreeBeats {
6790:   using F = function< Monoid(Monoid, Monoid) >;
6791:   using G = function< Monoid(Monoid, OperatorMonoid) >;
6792:   using H = function< OperatorMonoid(OperatorMonoid, OperatorMonoid) >;
6793:
6794:   int sz, height;
6795:   vector< Monoid > data;
6796:   vector< OperatorMonoid > lazy;
6797:   const F f;
6798:   const G g;
6799:   const H h;
6800:   const Monoid M1;
6801:   const OperatorMonoid OM0;
6802:
6803:
6804:   SegmentTreeBeats(int n, const F f, const G g, const H h,
6805:                    const Monoid &M1, const OperatorMonoid OM0)
6806:       : f(f), g(g), h(h), M1(M1), OM0(OM0) {
6807:     sz = 1;
6808:     height = 0;
6809:     while(sz < n) sz <<= 1, height++;
6810:     data.assign(2 * sz, M1);
6811:     lazy.assign(2 * sz, OM0);
6812:   }
6813:
6814:   void set(int k, const Monoid &x) {
6815:     data[k + sz] = x;
6816:   }
6817:
6818:   void build() {
6819:     for(int k = sz - 1; k > 0; k--) {
6820:       data[k] = f(data[2 * k + 0], data[2 * k + 1]);
6821:     }
6822:   }
6823:
6824:   inline void propagate(int k) {
6825:     if(lazy[k] != OM0) {
6826:       lazy[2 * k + 0] = h(lazy[2 * k + 0], lazy[k]);
6827:       lazy[2 * k + 1] = h(lazy[2 * k + 1], lazy[k]);
6828:       data[k] = reflect(k);
6829:       lazy[k] = OM0;
6830:     }
6831:   }
6832:
6833:   inline Monoid reflect(int k) {
6834:     return lazy[k] == OM0 ? data[k] : g(data[k], lazy[k]);
6835:   }
6836:
6837:   inline void recalc(int k) {
6838:     while(k >>= 1) data[k] = f(reflect(2 * k + 0), reflect(2 * k + 1));
6839:   }
6840:
6841:   inline void thrust(int k) {
6842:     for(int i = height; i > 0; i--) propagate(k >> i);
6843:   }
6844:
6845:   void update(int a, int b, const OperatorMonoid &x) {
6846:     thrust(a += sz);
6847:     thrust(b += sz - 1);
6848:     for(int l = a, r = b + 1; l < r; l >>= 1, r >>= 1) {
6849:       if(l & 1) lazy[l] = h(lazy[l], x), ++l;
6850:       if(r & 1) --r, lazy[r] = h(lazy[r], x);
6851:     }
```

```
 6852:     recalc(a);
 6853:     recalc(b);
 6854:   }
 6855:
 6856:   Monoid query(int a, int b) {
 6857:     thrust(a += sz);
 6858:     thrust(b += sz - 1);
 6859:     Monoid L = M1, R = M1;
 6860:     for(int l = a, r = b + 1; l < r; l >>= 1, r >>= 1) {
 6861:       if(l & 1) L = f(L, reflect(l++));
 6862:       if(r & 1) R = f(reflect(--r), R);
 6863:     }
 6864:     return f(L, R);
 6865:   }
 6866:
 6867:   Monoid operator[](const int &k) {
 6868:     return query(k, k + 1);
 6869:   }
 6870:
 6871:   template< typename Uku, typename Check, typename Func, typename X >
 6872:   void update_beats_subtree(int k, const X &x, const Uku &uku, const Check &check,
 const Func &func) {
 6873:     if(k >= sz) {
 6874:       auto v = reflect(k);
 6875:       if(uku(v, x)) return;
 6876:       if(check(v)) lazy[k] = func(v, x);
 6877:       return;
 6878:     }
 6879:     propagate(k);
 6880:     if(uku(data[k], x)) return;
 6881:     if(check(data[k])) {
 6882:       lazy[k] = func(data[k], x);
 6883:       return;
 6884:     }
 6885:     update_beats_subtree(k * 2 + 0, x, uku, check, func);
 6886:     update_beats_subtree(k * 2 + 1, x, uku, check, func);
 6887:     data[k] = f(reflect(2 * k + 0), reflect(2 * k + 1));
 6888:   }
 6889:
 6890:   template< typename Uku, typename Check, typename Func, typename X >
 6891:   void update_beats(int a, int b, const X &x, const Uku &uku, const Check &check,
const Func &func) {
 6892:     thrust(a += sz);
 6893:     thrust(b += sz - 1);
 6894:     for(int l = a, r = b + 1; l < r; l >>= 1, r >>= 1) {
 6895:       if(l & 1) update_beats_subtree(l++, x, uku, check, func);
 6896:       if(r & 1) update_beats_subtree(--r, x, uku, check, func);
 6897:     }
 6898:     recalc(a);
 6899:     recalc(b);
 6900:   }
 6901: };
 6902:
 6903:
 6904: ######################################################
 6905: #################### trie.cpp ####################
 6906: ######################################################
 6907:
 6908: template< int char_size >
 6909: struct TrieNode {
 6910:   int nxt[char_size];
 6911:
 6912:   int exist;
 6913:   vector< int > accept;
 6914:
 6915:   TrieNode() : exist(0) {
 6916:     memset(nxt, -1, sizeof(nxt));
 6917:   }
 6918: };
 6919:
```

```
6920: template< int char_size, int margin >
6921: struct Trie {
6922:   using Node = TrieNode< char_size >;
6923:
6924:   vector< Node > nodes;
6925:   int root;
6926:
6927:   Trie() : root(0) {
6928:     nodes.push_back(Node());
6929:   }
6930:
6931:   void update_direct(int node, int id) {
6932:     nodes[node].accept.push_back(id);
6933:   }
6934:
6935:   void update_child(int node, int child, int id) {
6936:     ++nodes[node].exist;
6937:   }
6938:
6939:   void add(const string &str, int str_index, int node_index, int id) {
6940:     if(str_index == str.size()) {
6941:       update_direct(node_index, id);
6942:     } else {
6943:       const int c = str[str_index] - margin;
6944:       if(nodes[node_index].nxt[c] == -1) {
6945:         nodes[node_index].nxt[c] = (int) nodes.size();
6946:         nodes.push_back(Node());
6947:       }
6948:       add(str, str_index + 1, nodes[node_index].nxt[c], id);
6949:       update_child(node_index, nodes[node_index].nxt[c], id);
6950:     }
6951:   }
6952:
6953:   void add(const string &str, int id) {
6954:     add(str, 0, 0, id);
6955:   }
6956:
6957:   void add(const string &str) {
6958:     add(str, nodes[0].exist);
6959:   }
6960:
6961:   void query(const string &str, const function< void(int) > &f, int str_index, int
node_index) {
6962:     for(auto &idx : nodes[node_index].accept) f(idx);
6963:     if(str_index == str.size()) {
6964:       return;
6965:     } else {
6966:       const int c = str[str_index] - margin;
6967:       if(nodes[node_index].nxt[c] == -1) return;
6968:       query(str, f, str_index + 1, nodes[node_index].nxt[c]);
6969:     }
6970:   }
6971:
6972:   void query(const string &str, const function< void(int) > &f) {
6973:     query(str, f, 0, 0);
6974:   }
6975:
6976:   int count() const {
6977:     return (nodes[0].exist);
6978:   }
6979:
6980:   int size() const {
6981:     return ((int) nodes.size());
6982:   }
6983: };
6984:
6985:
6986: ##################################################
6987: ############## union-find-undo.cpp ###############
6988: ##################################################
```

```
 6989:
 6990: struct UnionFindUndo {
 6991:   vector< int > data;
 6992:   stack< pair< int, int > > history;
 6993:
 6994:   UnionFindUndo(int sz) {
 6995:     data.assign(sz, -1);
 6996:   }
 6997:
 6998:   bool unite(int x, int y) {
 6999:     x = find(x), y = find(y);
 7000:     history.emplace(x, data[x]);
 7001:     history.emplace(y, data[y]);
 7002:     if(x == y) return (false);
 7003:     if(data[x] > data[y]) swap(x, y);
 7004:     data[x] += data[y];
 7005:     data[y] = x;
 7006:     return (true);
 7007:   }
 7008:
 7009:   int find(int k) {
 7010:     if(data[k] < 0) return (k);
 7011:     return (find(data[k]));
 7012:   }
 7013:
 7014:   int size(int k) {
 7015:     return (-data[find(k)]);
 7016:   }
 7017:
 7018:   void undo() {
 7019:     data[history.top().first] = history.top().second;
 7020:     history.pop();
 7021:     data[history.top().first] = history.top().second;
 7022:     history.pop();
 7023:   }
 7024:
 7025:   void snapshot() {
 7026:     while(history.size()) history.pop();
 7027:   }
 7028:
 7029:   void rollback() {
 7030:     while(history.size()) undo();
 7031:   }
 7032: };
 7033:
 7034:
 7035: ######################################################
 7036: ############## segment-tree-2d.cpp ###############
 7037: ######################################################
 7038:
 7039: template< typename structure_t, typename get_t, typename update_t >
 7040: struct SegmentTree2DCompressed {
 7041:
 7042:   using merge_f = function< get_t(get_t, get_t) >;
 7043:   using range_get_f = function< get_t(structure_t &, int, int) >;
 7044:   using update_f = function< void(structure_t &, int, update_t) >;
 7045:
 7046:   int sz;
 7047:   vector< structure_t > seg;
 7048:   const merge_f f;
 7049:   const range_get_f g;
 7050:   const update_f h;
 7051:   const get_t identity;
 7052:   vector< vector< int > > LL, RR;
 7053:   vector< vector< int > > beet;
 7054:
 7055:   SegmentTree2DCompressed(int n, const merge_f &f, const range_get_f &g, const upd
ate_f &h, const get_t &identity)
 7056:       : f(f), g(g), h(h), identity(identity) {
 7057:     sz = 1;
```

```
 7058:     while(sz < n) sz <<= 1;
 7059:     beet.resize(2 * sz);
 7060:     LL.resize(2 * sz);
 7061:     RR.resize(2 * sz);
 7062:   }
 7063:
 7064:   void update(int a, int x, update_t z, int k, int l, int r) {
 7065:     if(r <= a || a + 1 <= l) return;
 7066:     if(a <= l && r <= a + 1) return h(seg[k], x, z);
 7067:     update(a, LL[k][x], z, 2 * k + 0, l, (l + r) >> 1);
 7068:     update(a, RR[k][x], z, 2 * k + 1, (l + r) >> 1, r);
 7069:     return h(seg[k], x, z);
 7070:   }
 7071:
 7072:   void update(int x, int y, update_t z) {
 7073:     y = lower_bound(begin(beet[1]), end(beet[1]), y) - begin(beet[1]);
 7074:     return update(x, y, z, 1, 0, sz);
 7075:   }
 7076:
 7077:   get_t query(int a, int b, int x, int y, int k, int l, int r) {
 7078:     if(a >= r || b <= l) return identity;
 7079:     if(a <= l && r <= b) return g(seg[k], x, y);
 7080:     return f(query(a, b, LL[k][x], LL[k][y], 2 * k + 0, l, (l + r) >> 1),
 7081:              query(a, b, RR[k][x], RR[k][y], 2 * k + 1, (l + r) >> 1, r));
 7082:   }
 7083:
 7084:   get_t query(int a, int b, int x, int y) {
 7085:     x = lower_bound(begin(beet[1]), end(beet[1]), x) - begin(beet[1]);
 7086:     y = lower_bound(begin(beet[1]), end(beet[1]), y) - begin(beet[1]);
 7087:     return query(a, b, x, y, 1, 0, sz);
 7088:   }
 7089:
 7090:   void build() {
 7091:     for(int k = (int) beet.size() - 1; k >= sz; k--) {
 7092:       sort(begin(beet[k]), end(beet[k]));
 7093:       beet[k].erase(unique(begin(beet[k]), end(beet[k])), end(beet[k]));
 7094:     }
 7095:     for(int k = sz - 1; k > 0; k--) {
 7096:       beet[k].resize(beet[2 * k + 0].size() + beet[2 * k + 1].size());
 7097:       merge(begin(beet[2 * k + 0]), end(beet[2 * k + 0]), begin(beet[2 * k + 1]),
end(beet[2 * k + 1]), begin(beet[k]));
 7098:       beet[k].erase(unique(begin(beet[k]), end(beet[k])), end(beet[k]));
 7099:       LL[k].resize(beet[k].size() + 1);
 7100:       RR[k].resize(beet[k].size() + 1);
 7101:       int tail1 = 0, tail2 = 0;
 7102:       for(int i = 0; i < beet[k].size(); i++) {
 7103:         while(tail1 < beet[2 * k + 0].size() && beet[2 * k + 0][tail1] < beet[k][i
]) ++tail1;
 7104:         while(tail2 < beet[2 * k + 1].size() && beet[2 * k + 1][tail2] < beet[k][i
]) ++tail2;
 7105:         LL[k][i] = tail1, RR[k][i] = tail2;
 7106:       }
 7107:       LL[k][beet[k].size()] = (int) beet[2 * k + 0].size();
 7108:       RR[k][beet[k].size()] = (int) beet[2 * k + 1].size();
 7109:     }
 7110:     for(int k = 0; k < beet.size(); k++) {
 7111:       seg.emplace_back(structure_t(beet[k].size()));
 7112:     }
 7113:   }
 7114:
 7115:   void preupdate(int x, int y) {
 7116:     beet[x + sz].push_back(y);
 7117:   }
 7118: };
 7119:
 7120:
 7121: ##################################################
 7122: ############## union-rectangle.cpp ###############
 7123: ##################################################
 7124:
```

```
7125: template< typename T >
7126: struct UnionRectangle {
7127:   map< T, T > data;
7128:   int64 sum;
7129:
7130:   UnionRectangle() : sum(0) {
7131:     const T INF = numeric_limits< T >::max();
7132:     data[0] = INF;
7133:     data[INF] = 0;
7134:   }
7135:   void add_point(T x, T y) {
7136:     auto p = data.lower_bound(x);
7137:     if(p->second >= y) return;
7138:     const T nxtY = p->second;
7139:     --p;
7140:     while(p->second <= y) {
7141:       auto it = *p;
7142:       p = --data.erase(p);
7143:       sum -= (it.first - p->first) * (it.second - nxtY);
7144:     }
7145:     sum += (x - p->first) * (y - nxtY);
7146:     data[x] = y;
7147:   }
7148:
7149:   int64 get() {
7150:     return sum;
7151:   }
7152: };
7153:
7154:
7155: ######################################################
7156: ########## persistent-segment-tree.cpp ###########
7157: ######################################################
7158:
7159: template< typename Monoid >
7160: struct PersistentSegmentTree {
7161:   using F = function< Monoid(Monoid, Monoid) >;
7162:
7163:   struct Node {
7164:     Monoid data;
7165:     Node *l, *r;
7166:
7167:     Node(const Monoid &data) : data(data), l(nullptr), r(nullptr) {}
7168:   };
7169:
7170:
7171:   int sz;
7172:   const F f;
7173:   const Monoid M1;
7174:
7175:   PersistentSegmentTree(const F f, const Monoid &M1) : f(f), M1(M1) {}
7176:
7177:   Node *build(vector< Monoid > &v) {
7178:     sz = (int) v.size();
7179:     return build(0, (int) v.size(), v);
7180:   }
7181:
7182:   Node *merge(Node *l, Node *r) {
7183:     auto t = new Node(f(l->data, r->data));
7184:     t->l = l;
7185:     t->r = r;
7186:     return t;
7187:   }
7188:
7189:   Node *build(int l, int r, vector< Monoid > &v) {
7190:     if(l + 1 >= r) return new Node(v[l]);
7191:     return merge(build(l, (l + r) >> 1, v), build((l + r) >> 1, r, v));
7192:   }
7193:
7194:   Node *update(int a, const Monoid &x, Node *k, int l, int r) {
```

```
 7195:     if(r <= a || a + 1 <= l) {
 7196:       return k;
 7197:     } else if(a <= l && r <= a + 1) {
 7198:       return new Node(x);
 7199:     } else {
 7200:       return merge(update(a, x, k->l, l, (l + r) >> 1), update(a, x, k->r, (l + r)
 >> 1, r));
 7201:     }
 7202:   }
 7203:
 7204:   Node *update(Node *t, int k, const Monoid &x) {
 7205:     return update(k, x, t, 0, sz);
 7206:   }
 7207:
 7208:   Monoid query(int a, int b, Node *k, int l, int r) {
 7209:     if(r <= a || b <= l) {
 7210:       return M1;
 7211:     } else if(a <= l && r <= b) {
 7212:       return k->data;
 7213:     } else {
 7214:       return f(query(a, b, k->l, l, (l + r) >> 1),
 7215:                query(a, b, k->r, (l + r) >> 1, r));
 7216:     }
 7217:   }
 7218:
 7219:   Monoid query(Node *t, int a, int b) {
 7220:     return query(a, b, t, 0, sz);
 7221:   }
 7222: };
 7223:
 7224:
 7225: ######################################################
 7226: ############### red-black-tree.cpp ###############
 7227: ######################################################
 7228:
 7229: template< class T >
 7230: struct ArrayPool {
 7231:   vector< T > pool;
 7232:   vector< T * > stock;
 7233:   int ptr;
 7234:
 7235:   ArrayPool(int sz) : pool(sz), stock(sz) {}
 7236:
 7237:   inline T *alloc() { return stock[--ptr]; }
 7238:
 7239:   inline void free(T *t) { stock[ptr++] = t; }
 7240:
 7241:   void clear() {
 7242:     ptr = (int) pool.size();
 7243:     for(int i = 0; i < pool.size(); i++) stock[i] = &pool[i];
 7244:   }
 7245: };
 7246:
 7247: template< class D, class L, D (*f)(D, D), D (*g)(D, L), L (*h)(L, L), L (*p)(L, in
t) >
 7248: struct RedBlackTree {
 7249:   enum COLOR {
 7250:     BLACK, RED
 7251:   };
 7252:
 7253:   struct Node {
 7254:     Node *l, *r;
 7255:     COLOR color;
 7256:     int level, cnt;
 7257:     D key, sum;
 7258:     L lazy;
 7259:
 7260:     Node() {}
 7261:
 7262:     Node(const D &k, const L &laz) :
```

```
 7263:         key(k), sum(k), l(nullptr), r(nullptr), color(BLACK), level(0), cnt(1), la
zy(laz) {}
 7264:
 7265:     Node(Node *l, Node *r, const D &k, const L &laz) :
 7266:         key(k), color(RED), l(l), r(r), lazy(laz) {}
 7267:   };
 7268:
 7269:   ArrayPool< Node > pool;
 7270:
 7271:
 7272:   const D M1;
 7273:   const L OM0;
 7274:
 7275:   RedBlackTree(int sz, const D &M1, const L &OM0) :
 7276:       pool(sz), M1(M1), OM0(OM0) { pool.clear(); }
 7277:
 7278:
 7279:   inline Node *alloc(const D &key) {
 7280:     return &(*pool.alloc() = Node(key, OM0));
 7281:   }
 7282:
 7283:   inline Node *alloc(Node *l, Node *r) {
 7284:     auto t = &(*pool.alloc() = Node(l, r, M1, OM0));
 7285:     return update(t);
 7286:   }
 7287:
 7288:   virtual Node *clone(Node *t) { return t; }
 7289:
 7290:   inline int count(const Node *t) { return t ? t->cnt : 0; }
 7291:
 7292:   inline D sum(const Node *t) { return t ? t->sum : M1; }
 7293:
 7294:   Node *update(Node *t) {
 7295:     t->cnt = count(t->l) + count(t->r) + (!t->l || !t->r);
 7296:     t->level = t->l ? t->l->level + (t->l->color == BLACK) : 0;
 7297:     t->sum = f(f(sum(t->l), t->key), sum(t->r));
 7298:     return t;
 7299:   }
 7300:
 7301:   Node *propagate(Node *t) {
 7302:     t = clone(t);
 7303:     if(t->lazy != OM0) {
 7304:       if(!t->l) {
 7305:         t->key = g(t->key, p(t->lazy, 1));
 7306:       } else {
 7307:         if(t->l) {
 7308:           t->l = clone(t->l);
 7309:           t->l->lazy = h(t->l->lazy, t->lazy);
 7310:           t->l->sum = g(t->l->sum, p(t->lazy, count(t->l)));
 7311:         }
 7312:         if(t->r) {
 7313:           t->r = clone(t->r);
 7314:           t->r->lazy = h(t->r->lazy, t->lazy);
 7315:           t->r->sum = g(t->r->sum, p(t->lazy, count(t->r)));
 7316:         }
 7317:       }
 7318:       t->lazy = OM0;
 7319:     }
 7320:     return update(t);
 7321:   }
 7322:
 7323:   Node *rotate(Node *t, bool b) {
 7324:     t = propagate(t);
 7325:     Node *s;
 7326:     if(b) {
 7327:       s = propagate(t->l);
 7328:       t->l = s->r;
 7329:       s->r = t;
 7330:     } else {
 7331:       s = propagate(t->r);
```

```
7332:       t->r = s->l;
7333:       s->l = t;
7334:     }
7335:     update(t);
7336:     return update(s);
7337:   }
7338:
7339:   Node *submerge(Node *l, Node *r) {
7340:     if(l->level < r->level) {
7341:       r = propagate(r);
7342:       Node *c = (r->l = submerge(l, r->l));
7343:       if(r->color == BLACK && c->color == RED && c->l && c->l->color == RED) {
7344:         r->color = RED;
7345:         c->color = BLACK;
7346:         if(r->r->color == BLACK) return rotate(r, true);
7347:         r->r->color = BLACK;
7348:       }
7349:       return update(r);
7350:     }
7351:     if(l->level > r->level) {
7352:       l = propagate(l);
7353:       Node *c = (l->r = submerge(l->r, r));
7354:       if(l->color == BLACK && c->color == RED && c->r && c->r->color == RED) {
7355:         l->color = RED;
7356:         c->color = BLACK;
7357:         if(l->l->color == BLACK) return rotate(l, false);
7358:         l->l->color = BLACK;
7359:       }
7360:       return update(l);
7361:     }
7362:     return alloc(l, r);
7363:   }
7364:
7365:   Node *merge(Node *l, Node *r) {
7366:     if(!l || !r) return l ? l : r;
7367:     Node *c = submerge(l, r);
7368:     c->color = BLACK;
7369:     return c;
7370:   }
7371:
7372:   pair< Node *, Node * > split(Node *t, int k) {
7373:     if(!t) return {nullptr, nullptr};
7374:     t = propagate(t);
7375:     if(k == 0) return {nullptr, t};
7376:     if(k >= count(t)) return {t, nullptr};
7377:     Node *l = t->l, *r = t->r;
7378:     pool.free(t);
7379:     if(k < count(l)) {
7380:       auto pp = split(l, k);
7381:       return {pp.first, merge(pp.second, r)};
7382:     }
7383:     if(k > count(l)) {
7384:       auto pp = split(r, k - count(l));
7385:       return {merge(l, pp.first), pp.second};
7386:     }
7387:     return {l, r};
7388:   }
7389:
7390:   Node *build(int l, int r, const vector< D > &v) {
7391:     if(l + 1 >= r) return alloc(v[l]);
7392:     return merge(build(l, (l + r) >> 1, v), build((l + r) >> 1, r, v));
7393:   }
7394:
7395:   Node *build(const vector< D > &v) {
7396:     //pool.clear();
7397:     return build(0, (int) v.size(), v);
7398:   }
7399:
7400:   void dump(Node *r, typename vector< D >::iterator &it, L lazy) {
7401:     if(r->lazy != OM0) lazy = h(lazy, r->lazy);
```

```
7402:     if(!r->l || !r->r) {
7403:       *it++ = g(r->key, lazy);
7404:       return;
7405:     }
7406:     dump(r->l, it, lazy);
7407:     dump(r->r, it, lazy);
7408:   }
7409:
7410:   vector< D > dump(Node *r) {
7411:     vector< D > v((size_t) count(r));
7412:     auto it = begin(v);
7413:     dump(r, it, OM0);
7414:     return v;
7415:   }
7416:
7417:   string to_string(Node *r) {
7418:     auto s = dump(r);
7419:     string ret;
7420:     for(int i = 0; i < s.size(); i++) {
7421:       ret += std::to_string(s[i]);
7422:       ret += ", ";
7423:     }
7424:     return (ret);
7425:   }
7426:
7427:   void insert(Node *&t, int k, const D &v) {
7428:     auto x = split(t, k);
7429:     t = merge(merge(x.first, alloc(v)), x.second);
7430:   }
7431:
7432:   D erase(Node *&t, int k) {
7433:     auto x = split(t, k);
7434:     auto y = split(x.second, 1);
7435:     auto v = y.first->key;
7436:     pool.free(y.first);
7437:     t = merge(x.first, y.second);
7438:     return v;
7439:   }
7440:
7441:   D query(Node *&t, int a, int b) {
7442:     auto x = split(t, a);
7443:     auto y = split(x.second, b - a);
7444:     auto ret = sum(y.first);
7445:     t = merge(x.first, merge(y.first, y.second));
7446:     return ret;
7447:   }
7448:
7449:   void set_propagate(Node *&t, int a, int b, const L &pp) {
7450:     auto x = split(t, a);
7451:     auto y = split(x.second, b - a);
7452:     y.first->lazy = h(y.first->lazy, pp);
7453:     t = merge(x.first, merge(propagate(y.first), y.second));
7454:   }
7455:
7456:   void set_element(Node *&t, int k, const D &x) {
7457:     if(!t->l) {
7458:       t->key = t->sum = x;
7459:       return;
7460:     }
7461:     t = propagate(t);
7462:     if(k < count(t->l)) set_element(t->l, k, x);
7463:     else set_element(t->r, k - count(t->l), x);
7464:     t = update(t);
7465:   }
7466:
7467:   int size(Node *t) {
7468:     return count(t);
7469:   }
7470:
7471:   bool empty(Node *t) {
```

```
 7472:     return !t;
 7473:   }
 7474:
 7475:   Node *makeset() {
 7476:     return (nullptr);
 7477:   }
 7478: };
 7479:
 7480:
 7481: ######################################################
 7482: ####### randomized-binary-search-tree.cpp ########
 7483: ######################################################
 7484:
 7485: template< class Monoid, class OperatorMonoid = Monoid >
 7486: struct RandomizedBinarySearchTree {
 7487:   using F = function< Monoid(Monoid, Monoid) >;
 7488:   using G = function< Monoid(Monoid, OperatorMonoid) >;
 7489:   using H = function< OperatorMonoid(OperatorMonoid, OperatorMonoid) >;
 7490:   using P = function< OperatorMonoid(OperatorMonoid, int) >;
 7491:
 7492:   inline int xor128() {
 7493:     static int x = 123456789;
 7494:     static int y = 362436069;
 7495:     static int z = 521288629;
 7496:     static int w = 88675123;
 7497:     int t;
 7498:
 7499:     t = x ^ (x << 11);
 7500:     x = y;
 7501:     y = z;
 7502:     z = w;
 7503:     return w = (w ^ (w >> 19)) ^ (t ^ (t >> 8));
 7504:   }
 7505:
 7506:   struct Node {
 7507:     Node *l, *r;
 7508:     int cnt;
 7509:     Monoid key, sum;
 7510:     OperatorMonoid lazy;
 7511:
 7512:     Node() = default;
 7513:
 7514:     Node(const Monoid &k, const OperatorMonoid &p) : cnt(1), key(k), sum(k), lazy(
p), l(nullptr), r(nullptr) {}
 7515:   };
 7516:
 7517:   vector< Node > pool;
 7518:   int ptr;
 7519:
 7520:   const Monoid M1;
 7521:   const OperatorMonoid OM0;
 7522:   const F f;
 7523:   const G g;
 7524:   const H h;
 7525:   const P p;
 7526:
 7527:   RandomizedBinarySearchTree(int sz, const F &f, const Monoid &M1) :
 7528:       pool(sz), ptr(0), f(f), g(G()), h(H()), p(P()), M1(M1), OM0(OperatorMonoid()
) {}
 7529:
 7530:   RandomizedBinarySearchTree(int sz, const F &f, const G &g, const H &h, const P &
p,
 7531:                              const Monoid &M1, const OperatorMonoid &OM0) :
 7532:       pool(sz), ptr(0), f(f), g(g), h(h), p(p), M1(M1), OM0(OM0) {}
 7533:
 7534:   inline Node *alloc(const Monoid &key) { return &(pool[ptr++] = Node(key, OM0));
}
 7535:
 7536:   virtual Node *clone(Node *t) { return t; }
 7537:
```

```
7538:   inline int count(const Node *t) { return t ? t->cnt : 0; }
7539:
7540:   inline Monoid sum(const Node *t) { return t ? t->sum : M1; }
7541:
7542:   inline Node *update(Node *t) {
7543:     t->cnt = count(t->l) + count(t->r) + 1;
7544:     t->sum = f(f(sum(t->l), t->key), sum(t->r));
7545:     return t;
7546:   }
7547:
7548:   Node *propagate(Node *t) {
7549:     t = clone(t);
7550:     if(t->lazy != OM0) {
7551:       t->key = g(t->key, p(t->lazy, 1));
7552:       if(t->l) {
7553:         t->l = clone(t->l);
7554:         t->l->lazy = h(t->l->lazy, t->lazy);
7555:         t->l->sum = g(t->l->sum, p(t->lazy, count(t->l)));
7556:       }
7557:       if(t->r) {
7558:         t->r = clone(t->r);
7559:         t->r->lazy = h(t->r->lazy, t->lazy);
7560:         t->r->sum = g(t->r->sum, p(t->lazy, count(t->r)));
7561:       }
7562:       t->lazy = OM0;
7563:     }
7564:     return update(t);
7565:   }
7566:
7567:   Node *merge(Node *l, Node *r) {
7568:     if(!l || !r) return l ? l : r;
7569:     if(xor128() % (l->cnt + r->cnt) < l->cnt) {
7570:       l = propagate(l);
7571:       l->r = merge(l->r, r);
7572:       return update(l);
7573:     } else {
7574:       r = propagate(r);
7575:       r->l = merge(l, r->l);
7576:       return update(r);
7577:     }
7578:   }
7579:
7580:   pair< Node *, Node * > split(Node *t, int k) {
7581:     if(!t) return {t, t};
7582:     t = propagate(t);
7583:     if(k <= count(t->l)) {
7584:       auto s = split(t->l, k);
7585:       t->l = s.second;
7586:       return {s.first, update(t)};
7587:     } else {
7588:       auto s = split(t->r, k - count(t->l) - 1);
7589:       t->r = s.first;
7590:       return {update(t), s.second};
7591:     }
7592:   }
7593:
7594:   Node *build(int l, int r, const vector< Monoid > &v) {
7595:     if(l + 1 >= r) return alloc(v[l]);
7596:     return merge(build(l, (l + r) >> 1, v), build((l + r) >> 1, r, v));
7597:   }
7598:
7599:   Node *build(const vector< Monoid > &v) {
7600:     ptr = 0;
7601:     return build(0, (int) v.size(), v);
7602:   }
7603:
7604:   void dump(Node *r, typename vector< Monoid >::iterator &it) {
7605:     if(!r) return;
7606:     r = propagate(r);
7607:     dump(r->l, it);
```

```
7608:     *it = r->key;
7609:     dump(r->r, ++it);
7610:   }
7611:
7612:   vector< Monoid > dump(Node *r) {
7613:     vector< Monoid > v((size_t) count(r));
7614:     auto it = begin(v);
7615:     dump(r, it);
7616:     return v;
7617:   }
7618:
7619:   string to_string(Node *r) {
7620:     auto s = dump(r);
7621:     string ret;
7622:     for(int i = 0; i < s.size(); i++) ret += ", ";
7623:     return (ret);
7624:   }
7625:
7626:   void insert(Node *&t, int k, const Monoid &v) {
7627:     auto x = split(t, k);
7628:     t = merge(merge(x.first, alloc(v)), x.second);
7629:   }
7630:
7631:   void erase(Node *&t, int k) {
7632:     auto x = split(t, k);
7633:     t = merge(x.first, split(x.second, 1).second);
7634:   }
7635:
7636:   Monoid query(Node *&t, int a, int b) {
7637:     auto x = split(t, a);
7638:     auto y = split(x.second, b - a);
7639:     auto ret = sum(y.first);
7640:     t = merge(x.first, merge(y.first, y.second));
7641:     return ret;
7642:   }
7643:
7644:   void set_propagate(Node *&t, int a, int b, const OperatorMonoid &p) {
7645:     auto x = split(t, a);
7646:     auto y = split(x.second, b - a);
7647:     y.first->lazy = h(y.first->lazy, p);
7648:     t = merge(x.first, merge(propagate(y.first), y.second));
7649:   }
7650:
7651:   void set_element(Node *&t, int k, const Monoid &x) {
7652:     t = propagate(t);
7653:     if(k < count(t->l)) set_element(t->l, k, x);
7654:     else if(k == count(t->l)) t->key = t->sum = x;
7655:     else set_element(t->r, k - count(t->l) - 1, x);
7656:     t = update(t);
7657:   }
7658:
7659:
7660:   int size(Node *t) {
7661:     return count(t);
7662:   }
7663:
7664:   bool empty(Node *t) {
7665:     return !t;
7666:   }
7667:
7668:   Node *makeset() {
7669:     return nullptr;
7670:   }
7671: };
7672:
7673:
7674: ######################################################
7675: ############## bipartite-graph.cpp ###############
7676: ######################################################
7677:
```

```
7678: struct BipartiteGraph : UnionFind
7679: {
7680:   vector< int > color;
7681:
7682:   BipartiteGraph(int v) : color(v + v, -1), UnionFind(v + v) {}
7683:
7684:   bool bipartite_graph_coloring()
7685:   {
7686:     for(int i = 0; i < color.size() / 2; i++) {
7687:       int a = find(i);
7688:       int b = find(i + (int) color.size() / 2);
7689:       if(a == b) return (false);
7690:       if(color[a] < 0) color[a] = 0, color[b] = 1;
7691:     }
7692:     return (true);
7693:   }
7694:
7695:   bool operator[](int k)
7696:   {
7697:     return (bool(color[find(k)]));
7698:   }
7699: };
7700:
7701:
7702: ######################################################
7703: ################ li-chao-tree.cpp ################
7704: ######################################################
7705:
7706: template< typename T >
7707: struct LiChaoTree {
7708:   struct Line {
7709:     T a, b;
7710:
7711:     Line(T a, T b) : a(a), b(b) {}
7712:
7713:     inline T get(T x) const { return a * x + b; }
7714:
7715:     inline bool over(const Line &b, const T &x) const {
7716:       return get(x) < b.get(x);
7717:     }
7718:   };
7719:
7720:   vector< T > xs;
7721:   vector< Line > seg;
7722:   int sz;
7723:
7724:   LiChaoTree(const vector< T > &x, T INF) : xs(x) {
7725:     sz = 1;
7726:     while(sz < xs.size()) sz <<= 1;
7727:     while(xs.size() < sz) xs.push_back(xs.back() + 1);
7728:     seg.assign(2 * sz - 1, Line(0, INF));
7729:   }
7730:
7731:   void update(Line &x, int k, int l, int r) {
7732:     int mid = (l + r) >> 1;
7733:     auto latte = x.over(seg[k], xs[l]), malta = x.over(seg[k], xs[mid]);
7734:     if(malta) swap(seg[k], x);
7735:     if(l + 1 >= r) return;
7736:     else if(latte != malta) update(x, 2 * k + 1, l, mid);
7737:     else update(x, 2 * k + 2, mid, r);
7738:   }
7739:
7740:   void update(T a, T b) { // ax+b
7741:     Line l(a, b);
7742:     update(l, 0, 0, sz);
7743:   }
7744:
7745:   T query(int k) { // xs[k]
7746:     const T x = xs[k];
7747:     k += sz - 1;
```

```
 7748:     T ret = seg[k].get(x);
 7749:     while(k > 0) {
 7750:       k = (k - 1) >> 1;
 7751:       ret = min(ret, seg[k].get(x));
 7752:     }
 7753:     return ret;
 7754:   }
 7755: };
 7756:
 7757:
 7758: ######################################################
 7759: ########### persistent-binary-trie.cpp ###########
 7760: ######################################################
 7761:
 7762: template< typename T >
 7763: struct BinaryTrieNode {
 7764:   using Node = BinaryTrieNode< T >;
 7765:
 7766:   BinaryTrieNode< T > *nxt[2];
 7767:   int max_index;
 7768:
 7769:   BinaryTrieNode() : max_index(-1) {
 7770:     nxt[0] = nxt[1] = nullptr;
 7771:   }
 7772:
 7773:   void update_direct(int id) {
 7774:     max_index = max(max_index, id);
 7775:   }
 7776:
 7777:   void update_child(Node *child, int id) {
 7778:     max_index = max(max_index, id);
 7779:   }
 7780:
 7781:   Node *add(const T &bit, int bit_index, int id, bool need = true) {
 7782:     Node *node = need ? new Node(*this) : this;
 7783:     if(bit_index == -1) {
 7784:       node->update_direct(id);
 7785:     } else {
 7786:       const int c = (bit >> bit_index) & 1;
 7787:       if(node->nxt[c] == nullptr) node->nxt[c] = new Node(), need = false;
 7788:       node->nxt[c] = node->nxt[c]->add(bit, bit_index - 1, id, need);
 7789:       node->update_child(node->nxt[c], id);
 7790:     }
 7791:     return node;
 7792:   }
 7793:
 7794:   inline T min_query(T bit, int bit_index, int bit2, int l) {
 7795:     if(bit_index == -1) return bit;
 7796:     int c = (bit2 >> bit_index) & 1;
 7797:     if(nxt[c] != nullptr && l <= nxt[c]->max_index) {
 7798:       return nxt[c]->min_query(bit, bit_index - 1, bit2, l);
 7799:     } else {
 7800:       return nxt[1 ^ c]->min_query(bit | (1LL << bit_index), bit_index - 1, bit2,
l);
 7801:     }
 7802:   }
 7803: };
 7804:
 7805: template< typename T, int MAX_LOG >
 7806: struct PersistentBinaryTrie {
 7807:   using Node = BinaryTrieNode< T >;
 7808:   Node *root;
 7809:
 7810:   PersistentBinaryTrie(Node *root) : root(root) {}
 7811:
 7812:   PersistentBinaryTrie() : root(new Node()) {}
 7813:
 7814:   PersistentBinaryTrie add(const T &bit, int id) {
 7815:     return PersistentBinaryTrie(root->add(bit, MAX_LOG, id));
 7816:   }
```

```
  T min_query(int bit, int l) {
    return root->min_query(0, MAX_LOG, bit, l);
  }
};


################################################
################ segment-tree.cpp ################
################################################

template< typename Monoid >
struct SegmentTree {
  using F = function< Monoid(Monoid, Monoid) >;

  int sz;
  vector< Monoid > seg;

  const F f;
  const Monoid M1;

  SegmentTree(int n, const F f, const Monoid &M1) : f(f), M1(M1) {
    sz = 1;
    while(sz < n) sz <<= 1;
    seg.assign(2 * sz, M1);
  }

  void set(int k, const Monoid &x) {
    seg[k + sz] = x;
  }

  void build() {
    for(int k = sz - 1; k > 0; k--) {
      seg[k] = f(seg[2 * k + 0], seg[2 * k + 1]);
    }
  }

  void update(int k, const Monoid &x) {
    k += sz;
    seg[k] = x;
    while(k >>= 1) {
      seg[k] = f(seg[2 * k + 0], seg[2 * k + 1]);
    }
  }

  Monoid query(int a, int b) {
    Monoid L = M1, R = M1;
    for(a += sz, b += sz; a < b; a >>= 1, b >>= 1) {
      if(a & 1) L = f(L, seg[a++]);
      if(b & 1) R = f(seg[--b], R);
    }
    return f(L, R);
  }

  Monoid operator[](const int &k) const {
    return seg[k + sz];
  }

  template< typename C >
  int find_subtree(int a, const C &check, Monoid &M, bool type) {
    while(a < sz) {
      Monoid nxt = type ? f(seg[2 * a + type], M) : f(M, seg[2 * a + type]);
      if(check(nxt)) a = 2 * a + type;
      else M = nxt, a = 2 * a + 1 - type;
    }
    return a - sz;
  }


  template< typename C >
```

```
7887:   int find_first(int a, const C &check) {
7888:     Monoid L = M1;
7889:     if(a <= 0) {
7890:       if(check(f(L, seg[1]))) return find_subtree(1, check, L, false);
7891:       return -1;
7892:     }
7893:     int b = sz;
7894:     for(a += sz, b += sz; a < b; a >>= 1, b >>= 1) {
7895:       if(a & 1) {
7896:         Monoid nxt = f(L, seg[a]);
7897:         if(check(nxt)) return find_subtree(a, check, L, false);
7898:         L = nxt;
7899:         ++a;
7900:       }
7901:     }
7902:     return -1;
7903:   }
7904:
7905:   template< typename C >
7906:   int find_last(int b, const C &check) {
7907:     Monoid R = M1;
7908:     if(b >= sz) {
7909:       if(check(f(seg[1], R))) return find_subtree(1, check, R, true);
7910:       return -1;
7911:     }
7912:     int a = sz;
7913:     for(b += sz; a < b; a >>= 1, b >>= 1) {
7914:       if(b & 1) {
7915:         Monoid nxt = f(seg[--b], R);
7916:         if(check(nxt)) return find_subtree(b, check, R, true);
7917:         R = nxt;
7918:       }
7919:     }
7920:     return -1;
7921:   }
7922: };
7923:
7924:
7925:
7926: ######################################################
7927: ################ binary-trie.cpp #################
7928: ######################################################
7929:
7930: template< typename T, int MAX_LOG >
7931: struct BinaryTrie {
7932:   BinaryTrie *nxt[2];
7933:   T lazy;
7934:   int exist;
7935:   bool fill;
7936:   vector< int > accept;
7937:
7938:   BinaryTrie() : exist(0), lazy(0), nxt{nullptr, nullptr} {}
7939:
7940:   void add(const T &bit, int bit_index, int id) {
7941:     propagate(bit_index);
7942:     if(bit_index == -1) {
7943:       ++exist;
7944:       accept.push_back(id);
7945:     } else {
7946:       auto &to = nxt[(bit >> bit_index) & 1];
7947:       if(!to) to = new BinaryTrie();
7948:       to->add(bit, bit_index - 1, id);
7949:       ++exist;
7950:     }
7951:   }
7952:
7953:   void add(const T &bit, int id) {
7954:     add(bit, MAX_LOG, id);
7955:   }
7956:
```

```
7957:   void add(const T &bit) {
7958:     add(bit, exist);
7959:   }
7960:
7961:   void del(const T &bit, int bit_index) {
7962:     propagate(bit_index);
7963:     if(bit_index == -1) {
7964:       exist--;
7965:     } else {
7966:       nxt[(bit >> bit_index) & 1]->del(bit, bit_index - 1);
7967:       exist--;
7968:     }
7969:   }
7970:
7971:   void del(const T &bit) {
7972:     del(bit, MAX_LOG);
7973:   }
7974:
7975:
7976:   pair< T, BinaryTrie * > max_element(int bit_index) {
7977:     propagate(bit_index);
7978:     if(bit_index == -1) return {0, this};
7979:     if(nxt[1] && nxt[1]->size()) {
7980:       auto ret = nxt[1]->max_element(bit_index - 1);
7981:       ret.first |= T(1) << bit_index;
7982:       return ret;
7983:     } else {
7984:       return nxt[0]->max_element(bit_index - 1);
7985:     }
7986:   }
7987:
7988:   pair< T, BinaryTrie * > min_element(int bit_index) {
7989:     propagate(bit_index);
7990:     if(bit_index == -1) return {0, this};
7991:     if(nxt[0] && nxt[0]->size()) {
7992:       return nxt[0]->min_element(bit_index - 1);
7993:     } else {
7994:       auto ret = nxt[1]->min_element(bit_index - 1);
7995:       ret.first |= T(1) << bit_index;
7996:       return ret;
7997:     }
7998:   }
7999:
8000:   T mex_query(int bit_index) { // distinct-values
8001:     propagate(bit_index);
8002:     if(bit_index == -1 || !nxt[0]) return 0;
8003:     if(nxt[0]->size() == (T(1) << bit_index)) {
8004:       T ret = T(1) << bit_index;
8005:       if(nxt[1]) ret |= nxt[1]->mex_query(bit_index - 1);
8006:       return ret;
8007:     } else {
8008:       return nxt[0]->mex_query(bit_index - 1);
8009:     }
8010:   }
8011:
8012:   int64_t count_less(const T &bit, int bit_index) {
8013:     propagate(bit_index);
8014:     if(bit_index == -1) return 0;
8015:     int64_t ret = 0;
8016:     if((bit >> bit_index) & 1) {
8017:       if(nxt[0]) ret += nxt[0]->size();
8018:       if(nxt[1]) ret += nxt[1]->count_less(bit, bit_index - 1);
8019:     } else {
8020:       if(nxt[0]) ret += nxt[0]->count_less(bit, bit_index - 1);
8021:     }
8022:     return ret;
8023:   }
8024:
8025:   pair< T, BinaryTrie * > get_kth(int64_t k, int bit_index) { // 1-indexed
8026:     propagate(bit_index);
```

```
8027:     if(bit_index == -1) return {0, this};
8028:     if((nxt[0] ? nxt[0]->size() : 0) < k) {
8029:       auto ret = nxt[1]->get_kth(k - (nxt[0] ? nxt[0]->size() : 0), bit_index - 1)
;
8030:       ret.first |= T(1) << bit_index;
8031:       return ret;
8032:     } else {
8033:       return nxt[0]->get_kth(k, bit_index - 1);
8034:     }
8035:   }
8036:
8037:   pair< T, BinaryTrie * > max_element() {
8038:     assert(exist);
8039:     return max_element(MAX_LOG);
8040:   }
8041:
8042:   pair< T, BinaryTrie * > min_element() {
8043:     assert(exist);
8044:     return min_element(MAX_LOG);
8045:   }
8046:
8047:   T mex_query() {
8048:     return mex_query(MAX_LOG);
8049:   }
8050:
8051:   int size() const {
8052:     return exist;
8053:   }
8054:
8055:   void xorpush(const T &bit) {
8056:     lazy ^= bit;
8057:   }
8058:
8059:   int64_t count_less(const T &bit) {
8060:     return count_less(bit, MAX_LOG);
8061:   }
8062:
8063:   pair< T, BinaryTrie * > get_kth(int64_t k) {
8064:     assert(0 < k && k <= size());
8065:     return get_kth(k, MAX_LOG);
8066:   }
8067:
8068:   void propagate(int bit_index) {
8069:     if((lazy >> bit_index) & 1) swap(nxt[0], nxt[1]);
8070:     if(nxt[0]) nxt[0]->lazy ^= lazy;
8071:     if(nxt[1]) nxt[1]->lazy ^= lazy;
8072:     lazy = 0;
8073:   }
8074: };
8075:
8076:
8077: ######################################################
8078: ############ disjoint-sparse-table.cpp #############
8079: ######################################################
8080:
8081: template< typename Semigroup >
8082: struct DisjointSparseTable {
8083:   using F = function< Semigroup(Semigroup, Semigroup) >;
8084:   const F f;
8085:   vector< vector< Semigroup > > st;
8086:
8087:   DisjointSparseTable(const vector< Semigroup > &v, const F &f) : f(f) {
8088:     int b = 0;
8089:     while((1 << b) <= v.size()) ++b;
8090:     st.resize(b, vector< Semigroup >(v.size(), Semigroup()));
8091:     for(int i = 0; i < v.size(); i++) st[0][i] = v[i];
8092:     for(int i = 1; i < b; i++) {
8093:       int shift = 1 << i;
8094:       for(int j = 0; j < v.size(); j += shift << 1) {
8095:         int t = min(j + shift, (int) v.size());
```

```
8096:         st[i][t - 1] = v[t - 1];
8097:         for(int k = t - 2; k >= j; k--) st[i][k] = f(v[k], st[i][k + 1]);
8098:         if(v.size() <= t) break;
8099:         st[i][t] = v[t];
8100:         int r = min(t + shift, (int) v.size());
8101:         for(int k = t + 1; k < r; k++) st[i][k] = f(st[i][k - 1], v[k]);
8102:       }
8103:     }
8104:   }
8105:
8106:   Semigroup query(int l, int r) {
8107:     if(l >= --r) return st[0][l];
8108:     int p = 31 - __builtin_clz(l ^ r);
8109:     return f(st[p][l], st[p][r]);
8110:   }
8111: };
8112:
8113:
8114: ######################################################
8115: ############### wavelet-matrix.cpp ###############
8116: ######################################################
8117:
8118: struct SuccinctIndexableDictionary {
8119:   size_t length;
8120:   size_t blocks;
8121:   vector< unsigned > bit, sum;
8122:
8123:   SuccinctIndexableDictionary() {
8124:   }
8125:
8126:   SuccinctIndexableDictionary(size_t _length) {
8127:     length = _length;
8128:     blocks = (length + 31) >> 5;
8129:     bit.assign(blocks, 0U);
8130:     sum.assign(blocks, 0U);
8131:   }
8132:
8133:   void set(int k) {
8134:     bit[k >> 5] |= 1U << (k & 31);
8135:   }
8136:
8137:   void build() {
8138:     sum[0] = 0U;
8139:     for(int i = 1; i < blocks; i++) {
8140:       sum[i] = sum[i - 1] + __builtin_popcount(bit[i - 1]);
8141:     }
8142:   }
8143:
8144:   bool operator[](int k) const {
8145:     return (bool((bit[k >> 5] >> (k & 31)) & 1));
8146:   }
8147:
8148:   int rank(int k) {
8149:     return (sum[k >> 5] + __builtin_popcount(bit[k >> 5] & ((1U << (k & 31)) - 1))
);
8150:   }
8151:
8152:   int rank(bool val, int k) {
8153:     return (val ? rank(k) : k - rank(k));
8154:   }
8155:
8156:   int select(bool val, int k) {
8157:     if(k < 0 || rank(val, length) <= k) return (-1);
8158:     int low = 0, high = length;
8159:     while(high - low > 1) {
8160:       int mid = (low + high) >> 1;
8161:       if(rank(val, mid) >= k + 1) high = mid;
8162:       else low = mid;
8163:     }
8164:     return (high - 1);
```

```
 8165:   }
 8166:
 8167:   int select(bool val, int i, int l) {
 8168:     return select(val, i + rank(val, l));
 8169:   }
 8170: };
 8171:
 8172: template< class T, int MAXLOG >
 8173: struct WaveletMatrix {
 8174:   size_t length;
 8175:   SuccinctIndexableDictionary matrix[MAXLOG];
 8176:   int zs[MAXLOG];
 8177:   int buff1[MAXLOG], buff2[MAXLOG];
 8178:
 8179:   int freq_dfs(int d, int l, int r, T val, T a, T b) {
 8180:     if(l == r) return 0;
 8181:     if(d == MAXLOG) return (a <= val && val < b) ? r - l : 0;
 8182:     T nv = 1ULL << (MAXLOG - d - 1) | val, nnv = ((1ULL << (MAXLOG - d - 1)) - 1)
| nv;
 8183:     if(nnv < a || b <= val) return 0;
 8184:     if(a <= val && nnv < b) return r - l;
 8185:     int lc = matrix[d].rank(1, l), rc = matrix[d].rank(1, r);
 8186:     return freq_dfs(d + 1, l - lc, r - rc, val, a, b) +
 8187:            freq_dfs(d + 1, lc + zs[d], rc + zs[d], nv, a, b);
 8188:   }
 8189:
 8190:   WaveletMatrix(vector< T > data) {
 8191:     length = data.size();
 8192:     vector< T > l(length), r(length);
 8193:     for(int depth = 0; depth < MAXLOG; depth++) {
 8194:       matrix[depth] = SuccinctIndexableDictionary(length + 1);
 8195:       int left = 0, right = 0;
 8196:       for(int i = 0; i < length; i++) {
 8197:         bool k = (data[i] >> (MAXLOG - depth - 1)) & 1;
 8198:         if(k) r[right++] = data[i], matrix[depth].set(i);
 8199:         else l[left++] = data[i];
 8200:       }
 8201:       zs[depth] = left;
 8202:       matrix[depth].build();
 8203:       swap(l, data);
 8204:       for(int i = 0; i < right; i++) data[left + i] = r[i];
 8205:     }
 8206:   }
 8207:
 8208:   T access(int k) {
 8209:     int ret = 0;
 8210:     bool bit;
 8211:     for(int depth = 0; depth < MAXLOG; depth++) {
 8212:       bit = matrix[depth][k];
 8213:       ret = (ret << 1) | bit;
 8214:       k = matrix[depth].rank(bit, k) + zs[depth] * bit;
 8215:     }
 8216:     return (ret);
 8217:   }
 8218:
 8219:   int rank(T val, int k) {
 8220:     int l = 0, r = k;
 8221:     for(int depth = 0; depth < MAXLOG; depth++) {
 8222:       buff1[depth] = l, buff2[depth] = r;
 8223:       bool bit = (val >> (MAXLOG - depth - 1)) & 1;
 8224:       l = matrix[depth].rank(bit, l) + zs[depth] * bit;
 8225:       r = matrix[depth].rank(bit, r) + zs[depth] * bit;
 8226:     }
 8227:     return (r - l);
 8228:   }
 8229:
 8230:   int select(T val, int kth) {
 8231:     rank(val, length);
 8232:     for(int depth = MAXLOG - 1; depth >= 0; depth--) {
 8233:       bool bit = (val >> (MAXLOG - depth - 1)) & 1;
```

```
8234:       kth = matrix[depth].select(bit, kth, buff1[depth]);
8235:       if(kth >= buff2[depth] || kth < 0) return (-1);
8236:       kth -= buff1[depth];
8237:     }
8238:     return (kth);
8239:   }
8240:
8241:   int select(T val, int k, int l) {
8242:     return select(val, k + rank(val, l));
8243:   }
8244:
8245:   int quantile(int left, int right, int kth) {
8246:     if(right - left <= kth || kth < 0) return (-1);
8247:     T ret = 0;
8248:     for(int depth = 0; depth < MAXLOG; depth++) {
8249:       int l = matrix[depth].rank(1, left);
8250:       int r = matrix[depth].rank(1, right);
8251:       if(r - l > kth) {
8252:         left = l + zs[depth];
8253:         right = r + zs[depth];
8254:         ret |= 1ULL << (MAXLOG - depth - 1);
8255:       } else {
8256:         kth -= r - l;
8257:         left -= l;
8258:         right -= r;
8259:       }
8260:     }
8261:     return ret;
8262:   }
8263:
8264:   int rangefreq(int left, int right, T lower, T upper) {
8265:     return freq_dfs(0, left, right, 0, lower, upper);
8266:   }
8267: };
8268:
8269:
8270:
8271: ////////////////////////////////////////////////////
8272: ############# sqrt-decomposition.cpp #############
8273: ////////////////////////////////////////////////////
8274:
8275: template< typename T, typename E = int >
8276: struct SqrtDecomposition {
8277:
8278:   vector< E > block_add, elem_add;
8279:   vector< int > block_pos;
8280:   vector< T > data, lsum;
8281:   vector< vector< T > > sum;
8282:   int N, B, K;
8283:   E L;
8284:
8285:   SqrtDecomposition(int N, E L = 0) : N(N), L(L) { // find the sum of L or more in
the interval
8286:     B = (int) sqrt(N);
8287:     K = (N + B - 1) / B;
8288:
8289:     block_add.assign(K, 0);
8290:     block_pos.resize(N);
8291:     for(int k = 0; k < K; k++) {
8292:       for(int i = k * B; i < min((k + 1) * B, N); i++) block_pos[i] = k;
8293:     }
8294:     elem_add.assign(N, 0);
8295:     data.assign(N, 0);
8296:     sum.assign(K, vector< T >(B, 0));
8297:     lsum.assign(K, 0);
8298:   }
8299:
8300:
8301:   void build(const vector< E > &add, const vector< T > &dat) {
8302:     assert(add.size() == elem_add.size());
```

```
 8303:     assert(dat.size() == data.size());
 8304:     elem_add = add;
 8305:     data = dat;
 8306:     for(int k = 0; k < K; k++) {
 8307:       E tap = elem_add[k * B];
 8308:       for(int i = k * B; i < min((k + 1) * B, N); i++) tap = min(tap, elem_add[i])
;
 8309:       block_add[k] = tap;
 8310:       for(int i = k * B; i < min((k + 1) * B, N); i++) {
 8311:         elem_add[i] -= block_add[k];
 8312:         set(i, dat[i]);
 8313:       }
 8314:     }
 8315:   }
 8316:
 8317:   inline void del(int k) {
 8318:     sum[block_pos[k]][elem_add[k]] -= data[k];
 8319:     if(block_add[block_pos[k]] + elem_add[k] >= L) lsum[block_pos[k]] -= data[k];
 8320:   }
 8321:
 8322:   inline void set(int k) {
 8323:     while(sum[block_pos[k]].size() <= elem_add[k]) sum[block_pos[k]].push_back(0);
 8324:     sum[block_pos[k]][elem_add[k]] += data[k];
 8325:     if(block_add[block_pos[k]] + elem_add[k] >= L) lsum[block_pos[k]] += data[k];
 8326:   }
 8327:
 8328:   void set(int k, T x) {
 8329:     data[k] = x;
 8330:     set(k);
 8331:   }
 8332:
 8333:   void add(int a, int b) {
 8334:     for(int k = 0; k < K; k++) {
 8335:       int l = k * B;
 8336:       int r = min(l + B, N);
 8337:
 8338:       if(r <= a || b <= l) {
 8339:
 8340:       } else if(a <= l && r <= b) {
 8341:         block_add[k]++;
 8342:         if(0 <= L - block_add[k] && L - block_add[k] < sum[k].size()) {
 8343:           lsum[k] += sum[k][L - block_add[k]];
 8344:         }
 8345:       } else {
 8346:         for(int i = max(a, l); i < min(b, r); i++) {
 8347:           del(i);
 8348:           elem_add[i]++;
 8349:           set(i);
 8350:         }
 8351:       }
 8352:     }
 8353:   }
 8354:
 8355:
 8356:   void sub(int a, int b) {
 8357:     for(int k = 0; k < K; k++) {
 8358:       int l = k * B;
 8359:       int r = min(l + B, N);
 8360:
 8361:       if(r <= a || b <= l) {
 8362:
 8363:       } else if(a <= l && r <= b) {
 8364:         if(0 <= L - block_add[k] && L - block_add[k] < sum[k].size()) {
 8365:           lsum[k] -= sum[k][L - block_add[k]];
 8366:         }
 8367:         block_add[k]--;
 8368:       } else {
 8369:         if(0 <= L - block_add[k] && L - block_add[k] < sum[k].size()) {
 8370:           lsum[k] -= sum[k][L - block_add[k]];
 8371:         }
```

```
8372:         block_add[k]--;
8373:         for(int i = l; i < max(a, l); i++) {
8374:           del(i);
8375:           elem_add[i]++;
8376:           set(i);
8377:         }
8378:         for(int i = min(b, r); i < r; i++) {
8379:           del(i);
8380:           elem_add[i]++;
8381:           set(i);
8382:         }
8383:       }
8384:     }
8385:   }
8386:
8387:
8388:   T query(int a, int b, E x) {
8389:     T ret = 0;
8390:     for(int k = 0; k < K; k++) {
8391:       int l = k * B;
8392:       int r = min(l + B, N);
8393:
8394:       if(r <= a || b <= l) {
8395:
8396:       } else if(a <= l && r <= b) {
8397:         if(0 <= x - block_add[k] && x - block_add[k] < sum[k].size()) {
8398:           ret += sum[k][x - block_add[k]];
8399:         }
8400:       } else {
8401:         for(int i = max(a, l); i < min(b, r); i++) {
8402:           if(block_add[k] + elem_add[i] == x) ret += data[i];
8403:         }
8404:       }
8405:     }
8406:     return ret;
8407:   }
8408:
8409:
8410:   T query_low(int a, int b) {
8411:     T ret = 0;
8412:     for(int k = 0; k < K; k++) {
8413:       int l = k * B;
8414:       int r = min(l + B, N);
8415:
8416:       if(r <= a || b <= l) {
8417:
8418:       } else if(a <= l && r <= b) {
8419:         ret += lsum[k];
8420:       } else {
8421:         for(int i = max(a, l); i < min(b, r); i++) {
8422:           if(block_add[k] + elem_add[i] >= L) ret += data[i];
8423:         }
8424:       }
8425:     }
8426:     return ret;
8427:   }
8428: };
8429:
8430:
8431: ######################################################
8432: ############ priority-sum-structure.cpp ############
8433: ######################################################
8434:
8435: template< typename T, typename Compare = less< T >, typename RCompare = greater< T
> >
8436: struct PrioritySumStructure {
8437:
8438:   size_t k;
8439:   T sum;
8440:
```

```
  priority_queue< T, vector< T >, Compare > in, d_in;
  priority_queue< T, vector< T >, RCompare > out, d_out;

  PrioritySumStructure(int k) : k(k), sum(0) {}

  void modify() {
    while(in.size() - d_in.size() < k && !out.empty()) {
      auto p = out.top();
      out.pop();
      if(!d_out.empty() && p == d_out.top()) {
        d_out.pop();
      } else {
        sum += p;
        in.emplace(p);
      }
    }
    while(in.size() - d_in.size() > k) {
      auto p = in.top();
      in.pop();
      if(!d_in.empty() && p == d_in.top()) {
        d_in.pop();
      } else {
        sum -= p;
        out.emplace(p);
      }
    }
    while(!d_in.empty() && in.top() == d_in.top()) {
      in.pop();
      d_in.pop();
    }
  }

  T query() const {
    return sum;
  }

  void insert(T x) {
    in.emplace(x);
    sum += x;
    modify();
  }

  void erase(T x) {
    assert(size());
    if(!in.empty() && in.top() == x) {
      sum -= x;
      in.pop();
    } else if(!in.empty() && RCompare()(in.top(), x)) {
      sum -= x;
      d_in.emplace(x);
    } else {
      d_out.emplace(x);
    }
    modify();
  }

  void set_k(size_t kk) {
    k = kk;
    modify();
  }

  size_t get_k() const {
    return k;
  }

  size_t size() const {
    return in.size() + out.size() - d_in.size() - d_out.size();
  }
};

```

```
template< typename T >
using MaximumSum = PrioritySumStructure< T, greater< T >, less< T > >;

template< typename T >
using MinimumSum = PrioritySumStructure< T, less< T >, greater< T > >;


################################################
############### link-cut-tree.cpp ###############
################################################

template< typename Monoid = int, typename OperatorMonoid = Monoid >
struct LinkCutTree {
  using F = function< Monoid(Monoid, Monoid) >;
  using G = function< Monoid(Monoid, OperatorMonoid, int) >;
  using H = function< OperatorMonoid(OperatorMonoid, OperatorMonoid) >;
  using S = function< Monoid(Monoid) >;

  struct Node {
    Node *l, *r, *p;
    int idx;
    Monoid key, sum;
    OperatorMonoid lazy;

    bool rev;
    int sz;

    bool is_root() {
      return !p || (p->l != this && p->r != this);
    }

    Node(int idx, const Monoid &key, const OperatorMonoid &om) :
        idx(idx), key(key), sum(key), lazy(om), sz(1),
        l(nullptr), r(nullptr), p(nullptr), rev(false) {}
  };

  const Monoid M1;
  const OperatorMonoid OM0;
  const F f;
  const G g;
  const H h;
  const S s;

  LinkCutTree() : LinkCutTree([](Monoid a, Monoid b) { return a + b; }, [](Monoid
a) { return a; }, Monoid()) {}

  LinkCutTree(const F &f, const S &s, const Monoid &M1) :
      LinkCutTree(f, G(), H(), s, M1, OperatorMonoid()) {}

  LinkCutTree(const F &f, const G &g, const H &h, const S &s,
              const Monoid &M1, const OperatorMonoid &OM0) :
      f(f), g(g), h(h), s(s), M1(M1), OM0(OM0) {}

  Node *make_node(int idx, const Monoid &v = Monoid()) {
    return new Node(idx, v, OM0);
  }

  void propagate(Node *t, const OperatorMonoid &x) {
    t->lazy = h(t->lazy, x);
    t->key = g(t->key, x, 1);
    t->sum = g(t->sum, x, t->sz);
  }

  void toggle(Node *t) {
    assert(t);
    swap(t->l, t->r);
    t->sum = s(t->sum);
    t->rev ^= true;
  }
```

```
8580:
8581:   void push(Node *t) {
8582:     if(t->lazy != OM0) {
8583:       if(t->l) propagate(t->l, t->lazy);
8584:       if(t->r) propagate(t->r, t->lazy);
8585:       t->lazy = OM0;
8586:     }
8587:     if(t->rev) {
8588:       if(t->l) toggle(t->l);
8589:       if(t->r) toggle(t->r);
8590:       t->rev = false;
8591:     }
8592:   }
8593:
8594:   void update(Node *t) {
8595:     t->sz = 1;
8596:     t->sum = t->key;
8597:     if(t->l) t->sz += t->l->sz, t->sum = f(t->l->sum, t->sum);
8598:     if(t->r) t->sz += t->r->sz, t->sum = f(t->sum, t->r->sum);
8599:   }
8600:
8601:   void rotr(Node *t) {
8602:     auto *x = t->p, *y = x->p;
8603:     if((x->l = t->r)) t->r->p = x;
8604:     t->r = x, x->p = t;
8605:     update(x), update(t);
8606:     if((t->p = y)) {
8607:       if(y->l == x) y->l = t;
8608:       if(y->r == x) y->r = t;
8609:       update(y);
8610:     }
8611:   }
8612:
8613:   void rotl(Node *t) {
8614:     auto *x = t->p, *y = x->p;
8615:     if((x->r = t->l)) t->l->p = x;
8616:     t->l = x, x->p = t;
8617:     update(x), update(t);
8618:     if((t->p = y)) {
8619:       if(y->l == x) y->l = t;
8620:       if(y->r == x) y->r = t;
8621:       update(y);
8622:     }
8623:   }
8624:
8625:   void splay(Node *t) {
8626:     push(t);
8627:     while(!t->is_root()) {
8628:       auto *q = t->p;
8629:       if(q->is_root()) {
8630:         push(q), push(t);
8631:         if(q->l == t) rotr(t);
8632:         else rotl(t);
8633:       } else {
8634:         auto *r = q->p;
8635:         push(r), push(q), push(t);
8636:         if(r->l == q) {
8637:           if(q->l == t) rotr(q), rotr(t);
8638:           else rotl(t), rotr(t);
8639:         } else {
8640:           if(q->r == t) rotl(q), rotl(t);
8641:           else rotr(t), rotl(t);
8642:         }
8643:       }
8644:     }
8645:   }
8646:
8647:   Node *expose(Node *t) {
8648:     Node *rp = nullptr;
8649:     for(Node *cur = t; cur; cur = cur->p) {
```

```
8650:       splay(cur);
8651:       cur->r = rp;
8652:       update(cur);
8653:       rp = cur;
8654:     }
8655:     splay(t);
8656:     return rp;
8657:   }
8658:
8659:   void link(Node *child, Node *parent) {
8660:     expose(child);
8661:     expose(parent);
8662:     child->p = parent;
8663:     parent->r = child;
8664:     update(parent);
8665:   }
8666:
8667:   void cut(Node *child) {
8668:     expose(child);
8669:     auto *parent = child->l;
8670:     child->l = nullptr;
8671:     parent->p = nullptr;
8672:     update(child);
8673:   }
8674:
8675:   void evert(Node *t) {
8676:     expose(t);
8677:     toggle(t);
8678:     push(t);
8679:   }
8680:
8681:   Node *lca(Node *u, Node *v) {
8682:     if(get_root(u) != get_root(v)) return nullptr;
8683:     expose(u);
8684:     return expose(v);
8685:   }
8686:
8687:   vector< int > get_path(Node *x) {
8688:     vector< int > vs;
8689:     function< void(Node *) > dfs = [&](Node *cur) {
8690:       if(!cur) return;
8691:       push(cur);
8692:       dfs(cur->r);
8693:       vs.push_back(cur->idx);
8694:       dfs(cur->l);
8695:     };
8696:     expose(x);
8697:     dfs(x);
8698:     return vs;
8699:   }
8700:
8701:   void set_propagate(Node *t, const OperatorMonoid &x) {
8702:     expose(t);
8703:     propagate(t, x);
8704:     push(t);
8705:   }
8706:
8707:   Node *get_kth(Node *x, int k) {
8708:     expose(x);
8709:     while(x) {
8710:       push(x);
8711:       if(x->r && x->r->sz > k) {
8712:         x = x->r;
8713:       } else {
8714:         if(x->r) k -= x->r->sz;
8715:         if(k == 0) return x;
8716:         k -= 1;
8717:         x = x->l;
8718:       }
8719:     }
```

```
8720:     return nullptr;
8721:   }
8722:
8723:   Node *get_root(Node *x) {
8724:     expose(x);
8725:     while(x->l) {
8726:       push(x);
8727:       x = x->l;
8728:     }
8729:     return x;
8730:   }
8731: };
8732:
8733:
8734:
8735: ######################################################
8736: ########### persistent-union-find.cpp ############
8737: ######################################################
8738:
8739: struct PersistentUnionFind
8740: {
8741:   PersistentArray< int, 3 > data;
8742:
8743:   PersistentUnionFind() {}
8744:
8745:   PersistentUnionFind(int sz)
8746:   {
8747:     data.build(vector< int >(sz, -1));
8748:   }
8749:
8750:   int find(int k)
8751:   {
8752:     int p = data.get(k);
8753:     return p >= 0 ? find(p) : k;
8754:   }
8755:
8756:   int size(int k)
8757:   {
8758:     return (-data.get(find(k)));
8759:   }
8760:
8761:   PersistentUnionFind unite(int x, int y)
8762:   {
8763:     x = find(x);
8764:     y = find(y);
8765:     if(x == y) return *this;
8766:     auto u = data.get(x);
8767:     auto v = data.get(y);
8768:
8769:     if(u < v) {
8770:       auto a = data.mutable_get(x);
8771:       *a += v;
8772:       auto b = data.mutable_get(y);
8773:       *b = x;
8774:     } else {
8775:       auto a = data.mutable_get(y);
8776:       *a += u;
8777:       auto b = data.mutable_get(x);
8778:       *b = y;
8779:     }
8780:     return *this;
8781:   }
8782: };
8783:
8784:
```