Graphs

Topological sort

```
// Tri topologique
// O(|E|)
namespace Toposort {
  vector<bool> seen;
  vector<int> order;

  void dfs(const vector<vector<int>>& adj, int u) {
    seen[u] = true;
    for (int v: adj[u])
      if (!seen[v])
        dfs(adj, v);
    order.push_back(u);
  }

  vector<int> make(const vector<vector<int>> adj) {
    int n = (int)adj.size();
    seen = vector<bool>(n);
    order = vector<int>(0);
    for (int u = 0; u < n; ++u)
      if (!seen[u])
        dfs(adj, u);
    reverse(order.begin(), order.end());
    return order;
  }
};
```

Strongly connected components

```
// CFC
// O(|E|)
struct SCC {
  // scc[u] : CFC du noeud u
  vector<int> scc;
  // sccadj[i] : liste d'adjacence de la i-eme CFC
  vector<vector<int>> sccgraph;
  int nb_scc;

  vector<bool> seen;
  vector<vector<int>> adj;
  vector<vector<int>> adjt;

  void mark_scc(int u, int idscc) {
    scc[u] = idscc;
    seen[u] = true;
    for (int v: adjt[u])
      if (!seen[v])
        mark_scc(v, idscc);
  }

  void create_scc_graph(int u) {
    seen[u] = true;
    for (int v: adj[u]) {
      if (scc[v] != scc[u])
        sccgraph[scc[u]].push_back(scc[v]);
      if (!seen[v])
        create_scc_graph(v);
    }
  }

  SCC(vector<vector<int>> adj) {
    int n = (int)adj.size();
    this->adj = adj;
    adjt = vector<vector<int>>(n, vector<int>(0));
    for (int u = 0; u < n; ++u)
      for (int v: adj[u])
        adjt[v].push_back(u);

    vector<int> order = Toposort::make(adj);
    scc = vector<int>(n);
    nb_scc = 0;
    seen = vector<bool>(n);
    for (int t = n - 1; t >= 0; --t) {
      int u = order[t];
      if (!seen[u])
        mark_scc(u, nb_scc++);
    }

    // (optionnel) construction du DAG des CFC
    sccgraph = vector<vector<int>>(nb_scc, vector<int>(0));
    fill(seen.begin(), seen.end(), false);
    for (int u = 0; u < n; ++u)
      if (!seen[u])
        create_scc_graph(u);
  }
};
```

2-SAT

```
// 2-SAT
// O(n + m)
struct SAT2 {
  vector<bool> values;
  bool is_satisfiable;

  vector<int> order;
  vector<vector<int>> adj;
  int timer;

  // Formule a satisfaire :
  // (maxterms[0].fst ou maxterms[0].snd) et
  // (maxterms[1].fst ou maxterms[1].snd) et
  // ...
  // (maxterms[p].fst ou maxterms[p].snd)
  // litteral ::= 2*x pour x
  //              2*x+1 pour non(x)
  // 0 <= minterms[0].fst/snd < 2*m
  SAT2(vector<pair<int, int>> maxterms, int m) {
    int n = (int)maxterms.size();
    adj = vector<vector<int>>(2 * m, vector<int>(0));

    for (int i = 0; i < n; ++i) {
      adj[maxterms[i].first ^ 1].push_back(maxterms[i].second);
      adj[maxterms[i].second ^ 1].push_back(maxterms[i].first);
    }
```

```
    SCC scc(adj);
    is_satisfiable = true;
    for (int u = 0; u < 2 * m; u += 2)
      if (scc.scc[u] == scc.scc[u + 1])
        is_satisfiable = false;

    if (is_satisfiable) {
      values = vector<bool>(m);
      for (int u = 0; u < m; ++u)
        values[u] = scc.scc[2 * u] > scc.scc[2 * u + 1];
    }
  }
};
```

Hopcroft-Karp

```
// Nombre max de noeuds
const int MAXN = 50 * 1000;
// assert(INF > MAXN)
const int INF = 1000 * 1000 * 1000;

// Hopcroft-Karp
// Max cardinality matching en O(|E| sqrt(|V|))
struct HopcroftKarp {
  const int NIL = MAXN;
  vector<int> adj[MAXN + 1];
  int pairu[MAXN + 1];
  int pairv[MAXN + 1];
  int dist[MAXN + 1];
  int nl, nr;

  HopcroftKarp() {}
  // nl : #noeuds a gauche
  // nr : #noeuds a droite
  HopcroftKarp(int nl, int nr) : nl(nl), nr(nr) {}

  void add_edge(int u, int v) {
    adj[u].push_back(v);
  }

  bool bfs() {
    queue<int> q;

    for (int u = 0; u < nl; ++u) {
      if (pairu[u] == NIL) {
        dist[u] = 0;
        q.push(u);
      } else {
        dist[u] = INF;
      }
    }

    dist[NIL] = INF;
    while (!q.empty()) {
      int u = q.front();
      q.pop();

      if (dist[u] >= dist[NIL])
        continue;

      for (int v: adj[u]) {
        if (dist[pairv[v]] != INF) continue;
        dist[pairv[v]] = dist[u] + 1;
        q.push(pairv[v]);
      }
    }

    return dist[NIL] != INF;
  }

  bool dfs(int u) {
    if (u == NIL)
      return true;

    for (int v: adj[u]) {
      if (dist[pairv[v]] == dist[u] + 1 && dfs(pairv[v])) {
        pairv[v] = u;
        pairu[u] = v;
        return true;
      }
    }

    dist[u] = INF;
    return false;
  }

  int maxmatching() {
    fill(pairu, pairu + nl, NIL);
    fill(pairv, pairv + nr, NIL);
    int ans = 0;
    while (bfs())
      for (int u = 0; u < nl; ++u)
        if (pairu[u] == NIL && dfs(u))
          ++ans;
    return ans;
  }
};
```

Hungarian algorithm

```
// Inspire de http://e-maxx.ru/algo/assignment_hungary
// Algorithme hongrois
// Matching parfait de poids min
// Complexite : O(nm)
// Applications :
// - Max matching min weight
// - Max matching max weight (poids < 0)
// - Decomposer un DAG en un nombre min de chemins disjoints
// - Coloriage d'un arbre k-aire avec k couleurs, le coloriage de chaque noeud
// par une certaine couleur a un certain cout -> trouver un coloriage de cout
// min (dp[v][c] = cout min de colorier le sous-arbre en v sachant c(v) = c)
// - Etant donnee une matrice a[1..n][1..m], trouver deux tableaux u[1..n] et
// v[1..m] tq pour tout i,j : u[i] + v[j] <= a[i][j] et la somme des elements
// de u et v est max.
struct Hungarian {
  // p[i] = si 1 <= i <= m, vaut le noeud matche avec i (entre 1 et n)
  // vaut 0 si non matche
```

```
  vector<int> p;

  // a doit etre de taille (n + 1) x (m + 1), avec n <= m
  Hungarian(vector<vector<double>> a, int n, int m) {
    vector<double> u(n + 1), v(m + 1);
    vector<int> way(m + 1);
    p = vector<int>(m + 1);
    for (int i = 1; i <= n; ++i) {
      p[0] = i;
      int j0 = 0;
      vector<double> minv(m + 1, INF);
      vector<bool> used(m + 1, false);
      do {
        used[j0] = true;
        int i0 = p[j0], j1;
        double delta = LLINF;
        for (int j = 1; j <= m; ++j)
          if (!used[j]) {
            double cur = a[i0][j] - u[i0] - v[j];
            if (cur < minv[j]) {
              minv[j] = cur;
              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;
      } while (p[j0] != 0);
      do {
        int j1 = way[j0];
        p[j0] = p[j1];
        j0 = j1;
      } while (j0 != 0);
    }
  }
};
```

Dinic

```
// Nombre max de noeuds
const int MAXN = 10 * 1000;
// assert(INF > maxflow)
const ll INF = 1ll << 53;

// Dinic
// Flot max en O(V^2 * E)
struct Dinic {
  struct Edge { int u, v; ll cap, flow; };

  vector<int> adj[MAXN];
  vector<Edge> edges;
  int dist[MAXN];
  int idnext[MAXN];
  int n;

  Dinic() {}
  Dinic(int n) : n(n) {}

  // ajoute l'arete u -> v de capacite c
  void add_edge(int u, int v, int c) {
    edges.push_back({u, v, c, 0});
    adj[u].push_back((int)edges.size() - 1);
    edges.push_back({v, u, 0, 0});
    adj[v].push_back((int)edges.size() - 1);
  }

  bool bfs(int s, int t) {
    queue<int> q;
    fill(dist, dist + n, -1);
    dist[s] = 0;
    q.push(s);

    while (!q.empty()) {
      int u = q.front();
      q.pop();

      for (int e: adj[u]) {
        int v = edges[e].v;
        if (dist[v] == -1 && edges[e].flow < edges[e].cap) {
          dist[v] = dist[u] + 1;
          q.push(v);
        }
      }
    }

    return dist[t] != -1;
  }

  int dfs(int u, int t, ll flow) {
    if (flow == 0) return 0;
    if (u == t)    return flow;

    for (; idnext[u] < (int)adj[u].size(); ++idnext[u]) {
      int e = adj[u][idnext[u]];
      int v = edges[e].v;
      if (dist[v] != dist[u] + 1) continue;
      ll pushed = dfs(v, t, min(flow, edges[e].cap - edges[e].flow));
      if (pushed > 0) {
        edges[e].flow += pushed;
        edges[e ^ 1].flow -= pushed;
        return pushed;
      }
    }

    return 0;
  }

  // maxflow entre s et t
```

```
  ll maxflow(int s, int t) {
    ll ans = 0;
    while (bfs(s, t)) {
      fill(idnext, idnext + n, 0);
      ll pushed = 0;
      while ((pushed = dfs(s, t, INF)) > 0)
        ans += pushed;
    }
    return ans;
  }
};
```

Push-relabel

```
// Remarques sur max flow :
// - probleme avec une borne inf :
//  1) trouver un flot arbitraire entre S et T tq
//        cmin[u][v] <= flot[u][v] <= cmax[u][v]
// Ajouter une nouvelle source S' et un nouveau puits T'
// Poser c[u][v] := cmax[u][v] - cmin[u][v]
//       c[S'][v] := sum(cmin[u][v], u in V)
//       c[u][T'] := sum(cmin[u][v], v in V)
//       c[T][S] := INF
// (Theoreme) L'ancien graphe a un flot qui verifie les conditions ssi. le
// nouveau graphe a un flot saturant, ie. si sa valeur est exactement
//     sum(cmin[u][v], u, v in V)
// (et si c'est le cas, c'est forcement un flot max dans le nouveau graphe)
//  2) trouver le flot min verifiant ces conditions : dichotomie sur la valeur
//  de INF ?
const int INF = 1000 * 1000 * 1000;

// Max flow en O(V^3))
struct PushRelabel {
  // utiliser flow[u][v] a la fin de l'algo
  vector<vector<ll>> flow;

  // cap[1 .. n][1 .. n]
  // cap[u][v] = capacite entre les noeuds u et v
  // sur un graphe non complet, prendre cap[u][v] = 0 si not [(u, v) in E]
  // Calcule le flot max entre s et t avec les capacites cap
  PushRelabel(vector<vector<ll>> cap, int s, int t) {
    int n = cap.size();

    // sans perte de generalite
    for (int u = 0; u < n; ++u) cap[u][u] = 0;

    flow = vector<vector<ll>>(n, vector<ll>(n));
    vector<ll> e(n);
    vector<int> h(n);
    h[s] = n - 1;
    for (int i = 0; i < n; ++i) {
      flow[s][i] = cap[s][i];
      flow[i][s] = -flow[s][i];
      e[i] = cap[s][i];
    }

    vector<int> maxh(h);
    int sz = 0;
    while (true) {
      if (sz == 0)
        for (int i = 0; i < n; ++i)
          if (i != s && i != t && e[i] > 0) {
            if (sz > 0 && h[i] > h[maxh[0]]) sz = 0;
            if (sz == 0 || h[i] == h[maxh[0]]) maxh[sz++] = i;
          }

      if (sz == 0) break;

      while (sz > 0) {
        int i = maxh[sz - 1];
        bool pushed = false;
        for (int j = 0; j < n && e[i] > 0; ++j)
          if (cap[i][j] > flow[i][j] && h[i] == h[j] + 1) {
            pushed = true;
            ll addf = min(cap[i][j] - flow[i][j], e[i]);
            flow[i][j] += addf;
            flow[j][i] -= addf;
            e[i] -= addf;
            e[j] += addf;
            if (e[i] == 0) --sz;
          }

        if (!pushed) {
          h[i] = INF;
          for (int j = 0; j < n; ++j)
            if (cap[i][j] > flow[i][j] && h[i] > h[j] + 1)
              h[i] = h[j] + 1;
          if (h[i] > h[maxh[0]]) {
            sz = 0;
            break;
          }
        }
      }
    }
  }
};
```

Min-cost max-flow

```
// Tire de https://github.com/stjepang/snippets/blob/master/mcmf_dijkstra.cpp
// Min-cost max-flow (uses DFS)
//
// Given a directed weighted graph, source, and sink, computes the minimum cost
// of the maximum flow from source to sink.
// This version uses DFS to find shortest paths and gives good performance on
// very "shallow" graphs: graphs which have very short paths between source
// and sink (e.g. at most 10 edges).
// In such cases this algorithm can be orders of magnitude faster than the
// Dijkstra version.
//
// To use, call init(n), then add edges using edge(x, y, c, w), and finally
// call run(src, sink).
//
// Functions:
// - init(n) initializes the algorithm with the given number of nodes
// - edge(x, y, c, w) adds an edge x->y with capacity c and weight w
// - run(src, sink) runs the algorithm and returns {total_cost, total_flow}
//
```

```cpp
// Time complexity: O(V * E^3)
//
// Constants to configure:
// - MAXV is the maximum number of vertices
// - MAXE is the maximum number of edges (i.e. twice the calls to function edge)
// - oo is the "infinity" value
namespace Mcmf_dfs {
  const int MAXV = 1000100;
  const int MAXE = 1000100;
  const ll oo = 1e18;

  int V, E;
  int last[MAXV], curr[MAXV], bio[MAXV];
  ll pi[MAXV];
  int next[MAXE], adj[MAXE];
  ll cap[MAXE], cost[MAXE];

  void init(int n) {
    V = n;
    E = 0;
    fill(last, last + V, -1);
    fill(pi, pi + V, 0);
  }

  void edge(int x, int y, ll c, ll w) {
    adj[E] = y; cap[E] = c; cost[E] = +w; next[E] = last[x]; last[x] = E++;
    adj[E] = x; cap[E] = 0; cost[E] = -w; next[E] = last[y]; last[y] = E++;
  }

  ll push(int x, int sink, ll flow) {
    if (x == sink) return flow;
    if (bio[x]) return 0;
    bio[x] = true;

    for (int &e = curr[x]; e != -1; e = next[e]) {
      int y = adj[e];

      if (cap[e] && pi[x] == pi[y] + cost[e])
        if (ll f = push(y, sink, min(flow, cap[e])))
          return cap[e] -= f, cap[e ^ 1] += f, f;
    }
    return 0;
  }

  pair<ll, ll> run(int src, int sink) {
    ll total = 0;
    ll flow = 0;
    pi[src] = oo;

    for (;;) {
      fill(bio, bio + V, false);
      for (int i = 0; i < V; ++i) curr[i] = last[i];

      while (ll f = push(src, sink, oo)) {
        total += pi[src] * f;
        flow += f;
        fill(bio, bio + V, false);
      }

      ll inc = oo;
      for (int x = 0; x < V; ++x)
        if (bio[x])
          for (int e = last[x]; e != -1; e = next[e]) {
            int y = adj[e];
            if (cap[e] && !bio[y]) inc = min(inc, pi[y] + cost[e] - pi[x]);
          }

      if (inc == oo) break;

      for (int i = 0; i < V; ++i)
        if (bio[i])
          pi[i] += inc;
    }
    return {total, flow};
  }
}

// Min-cost max-flow (uses Dijkstra's algorithm)
//
// Given a directed weighted graph, source, and sink, computes the minimum cost
// of the maximum flow from source to sink.
// This version uses Dijkstra's algorithm and gives good performance on all
// kinds of graphs.
//
// To use, call init(n), then add edges using edge(x, y, c, w), and finally
// call run(src, sink).
//
// Functions:
// - init(n) initializes the algorithm with the given number of nodes
// - edge(x, y, c, w) adds an edge x->y with capacity c and weight w
// - run(src, sink) runs the algorithm and returns {total_cost, total_flow}
//
// Time complexity: O(V * E^2 log E)
//
// Constants to configure:
// - MAXV is the maximum number of vertices
// - MAXE is the maximum number of edges (i.e. twice the calls to function edge)
// - oo is the "infinity" value
namespace Mcmf_dijkstra {
  const int MAXV = 1000100;
  const int MAXE = 1000100;
  const ll oo = 1e18;

  int V, E;
  int last[MAXV], how[MAXV]; ll dist[MAXV];
  int next[MAXE], from[MAXE], adj[MAXE]; ll cap[MAXE], cost[MAXE];

  struct cmpf {
    bool operator () (int a, int b) {
      if (dist[a] != dist[b]) return dist[a] < dist[b];
      return a < b;
    }
  };
  set<int, cmpf> S;

  void init(int n) {
```

```cpp
    V = n;
    E = 0;
    fill(last, last + V, -1);
  }

  void edge(int x, int y, ll c, ll w) {
    from[E] = x; adj[E] = y; cap[E] = c; cost[E] = +w;
    next[E] = last[x]; last[x] = E++;
    from[E] = y; adj[E] = x; cap[E] = 0; cost[E] = -w;
    next[E] = last[y]; last[y] = E++;
  }

  pair<ll, ll> run(int src, int sink) {
    ll total = 0;
    ll flow = 0;

    for (;;) {
      fill(dist, dist + V, oo);
      dist[src] = 0;

      for (;;) {
        bool done = true;
        for (int x = 0; x < V; ++x)
          for (int e = last[x]; e != -1; e = next[e]) {
            if (cap[e] == 0) continue;

            int y = adj[e];
            ll val = dist[x] + cost[e];

            if (val < dist[y]) {
              dist[y] = val;
              how[y] = e;
              done = false;
            }
          }
        if (done) break;
      }

      if (dist[sink] >= oo / 2) break;

      ll aug = cap[how[sink]];
      for (int i = sink; i != src; i = from[how[i]])
        aug = min(aug, cap[how[i]]);

      for (int i = sink; i != src; i = from[how[i]]) {
        cap[how[i]] -= aug;
        cap[how[i] ^ 1] += aug;
        total += cost[how[i]] * aug;
      }
      flow += aug;
    }
    return {total, flow};
  }
}
```

Circulation

```cpp
// Tire de https://github.com/stjepang/snippets/blob/master/circulation.cpp
// Circulation
//
// Given a directed weighted graph, computes the minimum cost to run the maximum
// amount of circulation flow through the graph.
//
// Configure: MAXV
// Configure: MAXE (at least 2 * calls_to_edge)
//
// Functions:
// - init(n) initializes the algorithm with the given number of nodes
// - edge(x, y, c, w) adds an edge x->y with capacity c and weight w
// - run() runs the algorithm and returns total cost
//
// Time complexity: No idea, but it should be fast enough to solve any problem
// where V and E are up to around 1000.
//
// Constants to configure:
// - MAXV is the maximum number of vertices
// - MAXE is the maximum number of edges (i.e. twice the calls to function edge)

namespace Circu {
  const int MAXV = 1000100;
  const int MAXE = 1000100;

  int V, E;
  int how[MAXV], good[MAXV], bio[MAXV], cookie = 1; ll dist[MAXV];
  int from[MAXE], to[MAXE]; ll cap[MAXE], cost[MAXE];

  void init(int n) { V = n; E = 0; }

  void edge(int x, int y, ll c, ll w) {
    from[E] = x; to[E] = y; cap[E] = c; cost[E] = +w; ++E;
    from[E] = y; to[E] = x; cap[E] = 0; cost[E] = -w; ++E;
  }

  void reset() {
    fill(dist, dist + V, 0);
    fill(how, how + V, -1);
  }

  bool relax() {
    bool ret = false;
    for (int e = 0; e < E; ++e)
      if (cap[e]) {
        int x = from[e];
        int y = to[e];

        if (dist[x] + cost[e] < dist[y]) {
          dist[y] = dist[x] + cost[e];
          how[y] = e;
          ret = true;
        }
      }
    return ret;
  }

  ll cycle(int s, bool flip = false) {
    int x = s;
    ll c = cap[how[x]];
```

```
    do {
      int e = how[x];
      c = min(c, cap[e]);
      x = from[e];
    } while (x != s);

    ll sum = 0;
    do {
      int e = how[x];
      if (flip) {
        cap[e] -= c;
        cap[e ^ 1] += c;
      }
      sum += cost[e] * c;
      x = from[e];
    } while (x != s);
    return sum;
  }

  ll push(int x) {
    for (++cookie; bio[x] != cookie; x = from[how[x]]) {
      if (!good[x] || how[x] == -1 || cap[how[x]] == 0) return 0;
      bio[x] = cookie;
      good[x] = false;
    }
    return cycle(x) >= 0 ? 0 : cycle(x, true);
  }

  ll run() {
    reset();
    ll ret = 0;
    for (int step = 0; step < 2 * V; ++step) {
      if (step == V) reset();
      if (!relax()) continue;

      fill(good, good + V, true);
      for (int i = 0; i < V; ++i)
        if (ll w = push(i)) {
          ret += w;
          step = 0;
        }
    }
    return ret;
  }
}
```

Directed MST

```
// Tire de https://github.com/stjepang/snippets/blob/master/directed_mst.cpp
// Directed minimum spanning tree
//
// Given a directed weighted graph and root node, computes the minimum spanning
// directed tree (arborescence) on it.
//
// Complexity: O(N * M), where N is the number of nodes, and M the number
// of edges
struct Edge { int x, y, w; };

int dmst(int N, vector<Edge> E, int root) {
  const int oo = 1e9;

  vector<int> cost(N), back(N), label(N), bio(N);
  int ret = 0;

  for (;;) {
    fill(cost.begin(), cost.end(), oo);
    for (auto e : E) {
      if (e.x == e.y) continue;
      if (e.w < cost[e.y]) cost[e.y] = e.w, back[e.y] = e.x;
    }
    cost[root] = 0;

    for (int i = 0; i < N; ++i)
      if (cost[i] == oo)
        return -1;

    for (int i = 0; i < N; ++i)
      ret += cost[i];

    int K = 0;
    fill(label.begin(), label.end(), -1);
    fill(bio.begin(), bio.end(), -1);

    for (int i = 0; i < N; ++i) {
      int x = i;
      for (; x != root && bio[x] == -1; x = back[x]) bio[x] = i;

      if (x != root && bio[x] == i) {
        for (; label[x] == -1; x = back[x]) label[x] = K;
        ++K;
      }
    }
    if (K == 0) break;

    for (int i = 0; i < N; ++i)
      if (label[i] == -1)
        label[i] = K++;

    for (auto &e : E) {
      int xx = label[e.x];
      int yy = label[e.y];
      if (xx != yy) e.w -= cost[e.y];
      e.x = xx;
      e.y = yy;
    }

    root = label[root];
    N = K;
  }

  return ret;
}
```

Maths

Extended euclidian algorithm

```
// Euclide etendu
// Retourne PGCD(a, b)
// et u, v contiennent a l'issue de l'algo des bons couples de Bezout
// et |u| + |v| minimal, u <= v en cas d'egalite
ll gcd(ll a, ll b, ll& u, ll& v) {
  if (b == 0) {
    u = 1;
    v = 0;
    return a;
  }
  ll d = gcd(b, a % b, u, v);
  ll oldu = u;
  u = v;
  v = oldu - v * ll(a / b);
  return d;
}
```

Gauss

```
// Inspire de https://github.com/stjepang/snippets/blob/master/gauss.cpp
// Elimination de Gauss
// Resout un systeme d'equations lineaires
// Complexite : O(nb_lins * nb_cols^2)
// Si le systeme a au moins une solution, value contiendra une solution possible
const int MAX_NB_COLS = 250;
const double eps = 1e-8;

struct Gauss {
  // posnz[i] = -1 si la i-eme composante est libre
  int posnz[MAX_NB_COLS];
  // value[i] = la valeur de X(i) verifiant l'equation ci-dessous
  double value[MAX_NB_COLS];
  // vrai ssi. le systeme a >= 1 solution
  bool has_solution;

  // mat[0 .. nb_lins-1][0 .. nb_cols-1] * X = mat[0 .. nb_lins-1][nb_cols]
  Gauss(double mat[][MAX_NB_COLS + 1], int nb_lins, int nb_cols) {
    fill(posnz, posnz + nb_cols, -1);

    int posnz_cur = 0;
    for (int col = 0; col < nb_cols; ++col) {
      int max_lin = posnz_cur;

      for (int lin = max_lin + 1; lin < nb_lins; ++lin)
        if (fabs(mat[lin][col]) > fabs(mat[max_lin][col]))
          max_lin = lin;

      // La colonne est nulle
      // Condition de la forme == 0 si on est dans les entiers
      if (fabs(mat[max_lin][col]) < eps) continue;

      for (int i = 0; i <= nb_cols; ++i)
        swap(mat[max_lin][i], mat[posnz_cur][i]);

      for (int lin = 0; lin < nb_lins; ++lin) {
        if (lin == posnz_cur) continue;
        // Pour Gauss modulaire : remplacer par l'inverse de mat[posnz_cur][col]
        double factor = mat[lin][col] / mat[posnz_cur][col];
        for (int i = 0; i <= nb_cols; ++i)
          mat[lin][i] -= factor * mat[posnz_cur][i];
          // Gauss mod : rajouter le modulo
      }

      posnz[col] = posnz_cur++;
    }

    // Genere une solution valide
    for (int col = 0; col < nb_cols; ++col) {
      if (posnz[col] != -1)
        value[col] = mat[posnz[col]][nb_cols] / mat[posnz[col]][col];
      // Gauss mod
      else
        value[col] = 0;
    }

    // Verifie que la solution generee est valide
    has_solution = true;
    for (int lin = 0; lin < nb_lins; ++lin) {
      double sum = 0;
      for (int col = 0; col < nb_cols; ++col)
        sum += mat[lin][col] * value[col]; // Gauss mod
      if (fabs(sum - mat[lin][nb_cols]) > eps) // Gauss mod
        has_solution = false;
    }
  }
};
```

Prime list

```
// Calcule la liste des nombres premiers dans [2, n]
// Stocke egalement pour tout i lp[i] le plus petit diviseur premier de i
// (permet de factoriser en temps lineaire)
// Complexite : O(n)
struct PrimeList {
  // primes[i] = i-eme premier de [2, n]
  vector<int> primes;
  // lp[i] = plus petit diviseur premier de i
  vector<int> lp;

  PrimeList(int n) {
    lp = vector<int>(n + 1);
    for (int i = 2; i <= n; ++i) {
      if (lp[i] == 0) {
        lp[i] = i;
        primes.push_back(i);
      }
      for (int j = 0; j < (int)primes.size() && primes[j] <= lp[i]
                      && i * primes[j] <= n; ++j)
        lp[i * primes[j]] = primes[j];
    }
  }

  // n > 0
  // retourne la liste des facteurs premiers de n
```

```
  // (ex: factorize(12) = [2, 2, 3])
  vector<int> factorize(int n) {
    vector<int> ans;
    while (n > 1) {
      ans.push_back(lp[n]);
      n /= lp[n];
    }
    return ans;
  }
};
```

Factorization

```
// factorise n en produit de ses facteurs premiers
// Complexite : O(sqrt(n))
vector<int> factorize(int n) {
  vector<int> ans;
  for (int i = 2; i * i <= n; ++i)
    while (n % i == 0) {
      ans.push_back(i);
      n /= i;
    }
  if (n > 1)
    ans.push_back(n);
  return ans;
}
```

FFT

```
// From https://github.com/stjepang/snippets
//
// Fast Fourier transform
//
// Caling mult(a, b, c, len) is identical to:
//   REP(i, 2*len) tmp[i] = 0
//   REP(i, len) REP(j, len) tmp[i+j] += a[i] * b[j];
//   REP(i, 2*len) c[i] = tmp[i];
//
// There is also a variant with modular arithmetic: mult_mod.
//
// Common use cases:
// - big integer multiplication
// - convolutions in dynamic programming
//
// Time complexity: O(N log N), where N is the length of arrays
//
// Constants to configure:
// - MAX must be at least 2^ceil(log2(2 * len))
#define MY_PI     3.14159265358979323846
#define REP(i, n) for (int i = 0; i < (n); ++i)

namespace FFT {
  const int MAX = 1 << 20;

  typedef ll value;
  typedef complex<double> comp;

  int N;
  comp omega[MAX];
  comp a1[MAX], a2[MAX];
  comp z1[MAX], z2[MAX];

  void fft(comp *a, comp *z, int m = N) {
    if (m == 1) {
      z[0] = a[0];
    } else {
      int s = N / m;
      m /= 2;

      fft(a, z, m);
      fft(a + s, z + m, m);

      for (int i = 0; i < m; ++i) {
        comp c = omega[s * i] * z[m + i];
        z[m + i] = z[i] - c;
        z[i] += c;
      }
    }
  }

  // len = longueur de a et b
  // c = a * b
  void mult(value *a, value *b, value *c, int len) {
    N = 2 * len;

    while (N & (N - 1))
      ++N;

    assert(N <= MAX);
    fill(a1, a1 + N, 0);
    fill(a2, a2 + N, 0);
    for (int i = 0; i < len; ++i) {
      a1[i] = a[i];
      a2[i] = b[i];
    }

    for (int i = 0; i < N; ++i)
      omega[i] = polar(1., 2 * MY_PI / N * i);

    fft(a1, z1, N);
    fft(a2, z2, N);

    for (int i = 0; i < N; ++i) {
      omega[i] = comp(1, 0) / omega[i];
      a1[i] = z1[i] * z2[i] / comp(N, 0);
    }

    fft(a1, z1, N);
    for (int i = 0; i < 2 * len; ++i)
      c[i] = round(z1[i].real());
  }

  // len = longueur de a et b
  // c = a * b [mod]
  void mult_mod(value *a, value *b, value *c, int len, int mod) {
    static value a0[MAX], a1[MAX];
    static value b0[MAX], b1[MAX];
```

```
    static value c0[MAX], c1[MAX], c2[MAX];

    for (int i = 0; i < len; ++i) {
      a0[i] = a[i] & 0xFFFF;
      a1[i] = a[i] >> 16;
      b0[i] = b[i] & 0xFFFF;
      b1[i] = b[i] >> 16;
    }

    FFT::mult(a0, b0, c0, len);
    FFT::mult(a1, b1, c2, len);

    for (int i = 0; i < len; ++i) {
      a0[i] += a1[i];
      b0[i] += b1[i];
    }

    FFT::mult(a0, b0, c1, len);

    for (int i = 0; i < 2 * len; ++i) {
      c1[i] -= c0[i] + c2[i];
      c1[i] %= mod;
      c2[i] %= mod;
      c[i] = (c0[i] + (c1[i] << 16) + (c2[i] << 32)) % mod;
    }
  }
}
```

Simplex

```
// Simplexe
// Tire de https://github.com/jaehyunp/stanfordacm/
typedef long double DOUBLE;
typedef vector<DOUBLE> VD;
typedef vector<VD> VVD;
typedef vector<int> VI;
const DOUBLE EPS = 1e-9;

struct LPSolver {
  int m, n;
  VI B, N;
  VVD D;

  LPSolver(const VVD &A, const VD &b, const VD &c) :
    m(b.size()), n(c.size()), N(n + 1), B(m), D(m + 2, VD(n + 2)) {
    for (int i = 0; i < m; i++)
      for (int j = 0; j < n; j++)
        D[i][j] = A[i][j];
    for (int i = 0; i < m; i++) {
      B[i] = n + i;
      D[i][n] = -1;
      D[i][n + 1] = b[i];
    }
    for (int j = 0; j < n; j++) {
      N[j] = j;
      D[m][j] = -c[j];
    }
    N[n] = -1; D[m + 1][n] = 1;
  }

  void Pivot(int r, int s) {
    for (int i = 0; i < m + 2; i++) if (i != r)
      for (int j = 0; j < n + 2; j++) if (j != s)
        D[i][j] -= D[r][j] * D[i][s] / D[r][s];
    for (int j = 0; j < n + 2; j++) if (j != s) D[r][j] /= D[r][s];
    for (int i = 0; i < m + 2; i++) if (i != r) D[i][s] /= -D[r][s];
    D[r][s] = 1.0 / D[r][s];
    swap(B[r], N[s]);
  }

  bool Simplex(int phase) {
    int x = phase == 1 ? m + 1 : m;
    while (true) {
      int s = -1;
      for (int j = 0; j <= n; j++) {
        if (phase == 2 && N[j] == -1) continue;
        if (s == -1 || D[x][j] < D[x][s] || D[x][j] == D[x][s] && N[j] < N[s])
          s = j;
      }
      if (D[x][s] > -EPS) return true;
      int r = -1;
      for (int i = 0; i < m; i++) {
        if (D[i][s] < EPS) continue;
        if (r == -1 || D[i][n + 1] / D[i][s] < D[r][n + 1] / D[r][s] ||
          (D[i][n + 1] / D[i][s]) == (D[r][n + 1] / D[r][s]) && B[i] < B[r])
          r = i;
      }
      if (r == -1) return false;
      Pivot(r, s);
    }
  }

  DOUBLE Solve(VD &x) {
    int r = 0;
    for (int i = 1; i < m; i++)
      if (D[i][n + 1] < D[r][n + 1])
        r = i;
    if (D[r][n + 1] < -EPS) {
      Pivot(r, n);
      if (!Simplex(1) || D[m + 1][n + 1] < -EPS)
        return -numeric_limits<DOUBLE>::infinity();
      for (int i = 0; i < m; i++) if (B[i] == -1) {
        int s = -1;
        for (int j = 0; j <= n; j++)
          if (s == -1 || D[i][j] < D[i][s] || D[i][j] == D[i][s] && N[j] < N[s])
            s = j;
        Pivot(i, s);
      }
    }
    if (!Simplex(2)) return numeric_limits<DOUBLE>::infinity();
    x = VD(n);
    for (int i = 0; i < m; i++) if (B[i] < n) x[B[i]] = D[i][n + 1];
    return D[m][n + 1];
  }
};
```

Geometry

```cpp
// Geometrie
// Tire de https://github.com/jaehyunp/stanfordacm/
double INF = 1e100;
double EPS = 1e-12;

struct PT {
  double x, y;
  PT() {}
  PT(double x, double y) : x(x), y(y) {}
  PT(const PT &p) : x(p.x), y(p.y)    {}
  PT operator + (const PT &p) const { return PT(x + p.x, y + p.y); }
  PT operator - (const PT &p) const { return PT(x - p.x, y - p.y); }
  PT operator * (double c) const { return PT(x * c, y * c); }
  PT operator / (double c) const { return PT(x / c, y / c); }
  void print() const { printf("(%f, %f)", x, y); }
};

double dot(PT p, PT q) { return p.x * q.x + p.y * q.y; }
double dist2(PT p, PT q) { return dot(p - q, p - q); }
double cross(PT p, PT q) { return p.x * q.y - p.y * q.x; }

// rotate a point CCW or CW around the origin
PT RotateCCW90(PT p) { return PT(-p.y, p.x); }
PT RotateCW90(PT p) { return PT(p.y, -p.x); }
PT RotateCCW(PT p, double t) {
  return PT(p.x * cos(t) - p.y * sin(t), p.x * sin(t) + p.y * cos(t));
}

// project point c onto line through a and b
// assuming a != b
PT ProjectPointLine(PT a, PT b, PT c) {
  return a + (b - a) * dot(c - a, b - a) / dot(b - a, b - a);
}

// project point c onto line segment through a and b
PT ProjectPointSegment(PT a, PT b, PT c) {
  double r = dot(b - a, b - a);
  if (fabs(r) < EPS) return a;
  r = dot(c - a, b - a) / r;
  if (r < 0) return a;
  if (r > 1) return b;
  return a + (b - a) * r;
}

// compute distance from c to segment between a and b
double DistancePointSegment(PT a, PT b, PT c) {
  return sqrt(dist2(c, ProjectPointSegment(a, b, c)));
}

// compute distance between point (x,y,z) and plane ax+by+cz=d
double DistancePointPlane(double x, double y, double z,
                          double a, double b, double c, double d)
{
  return fabs(a * x + b * y + c * z - d) / sqrt(a * a + b * b + c * c);
}

// determine if lines from a to b and c to d are parallel or collinear
bool LinesParallel(PT a, PT b, PT c, PT d) {
  return fabs(cross(b - a, c - d)) < EPS;
}

bool LinesCollinear(PT a, PT b, PT c, PT d) {
  return LinesParallel(a, b, c, d)
      && fabs(cross(a - b, a - c)) < EPS
      && fabs(cross(c - d, c - a)) < EPS;
}

// determine if line segment from a to b intersects with
// line segment from c to d
bool SegmentsIntersect(PT a, PT b, PT c, PT d) {
  if (LinesCollinear(a, b, c, d)) {
    if (dist2(a, c) < EPS || dist2(a, d) < EPS ||
      dist2(b, c) < EPS || dist2(b, d) < EPS) return true;
    if (dot(c - a, c - b) > 0 && dot(d - a, d - b) > 0 && dot(c - b, d - b) > 0)
      return false;
    return true;
  }
  if (cross(d - a, b - a) * cross(c - a, b - a) > 0) return false;
  if (cross(a - c, d - c) * cross(b - c, d - c) > 0) return false;
  return true;
}

// compute intersection of line passing through a and b
// with line passing through c and d, assuming that unique
// intersection exists; for segment intersection, check if
// segments intersect first
PT ComputeLineIntersection(PT a, PT b, PT c, PT d) {
  b = b - a; d = c - d; c = c - a;
  assert(dot(b, b) > EPS && dot(d, d) > EPS);
  return a + b * cross(c, d) / cross(b, d);
}

// compute center of circle given three points
PT ComputeCircleCenter(PT a, PT b, PT c) {
  b = (a + b) / 2;
  c = (a + c) /2;
  return ComputeLineIntersection(b, b + RotateCW90(a - b), c,
                                 c + RotateCW90(a - c));
}

// determine if point is in a possibly non-convex polygon (by William
// Randolph Franklin); returns 1 for strictly interior points, 0 for
// strictly exterior points, and 0 or 1 for the remaining points.
// Note that it is possible to convert this into an *exact* test using
// integer arithmetic by taking care of the division appropriately
// (making sure to deal with signs properly) and then by writing exact
// tests for checking point on polygon boundary
bool PointInPolygon(const vector<PT> &p, PT q) {
  bool c = 0;
  for (int i = 0; i < p.size(); i++) {
    int j = (i + 1) % p.size();
    if ((p[i].y <= q.y && q.y < p[j].y ||
      p[j].y <= q.y && q.y < p[i].y) &&
      q.x < p[i].x + (p[j].x - p[i].x) * (q.y - p[i].y) / (p[j].y - p[i].y))
      c = !c;
  }
```

```cpp
  return c;
}

// determine if point is on the boundary of a polygon
bool PointOnPolygon(const vector<PT> &p, PT q) {
  for (int i = 0; i < p.size(); i++)
    if (dist2(ProjectPointSegment(p[i], p[(i + 1) % p.size()], q), q) < EPS)
      return true;
  return false;
}

// compute intersection of line through points a and b with
// circle centered at c with radius r > 0
vector<PT> CircleLineIntersection(PT a, PT b, PT c, double r) {
  vector<PT> ret;
  b = b - a;
  a = a - c;
  double A = dot(b, b);
  double B = dot(a, b);
  double C = dot(a, a) - r * r;
  double D = B * B - A * C;
  if (D < -EPS) return ret;
  ret.push_back(c + a + b * (-B + sqrt(D + EPS)) / A);
  if (D > EPS)
    ret.push_back(c + a + b * (-B - sqrt(D)) / A);
  return ret;
}

// compute intersection of circle centered at a with radius r
// with circle centered at b with radius R
vector<PT> CircleCircleIntersection(PT a, PT b, double r, double R) {
  vector<PT> ret;
  double d = sqrt(dist2(a, b));
  if (d > r + R || d + min(r, R) < max(r, R)) return ret;
  double x = (d * d - R * R + r * r) / (2 * d);
  double y = sqrt(r * r - x * x);
  PT v = (b - a)/d;
  ret.push_back(a + v * x + RotateCCW90(v) * y);
  if (y > 0)
    ret.push_back(a + v * x - RotateCCW90(v) * y);
  return ret;
}

// This code computes the area or centroid of a (possibly nonconvex)
// polygon, assuming that the coordinates are listed in a clockwise or
// counterclockwise fashion.  Note that the centroid is often known as
// the "center of gravity" or "center of mass".
double ComputeSignedArea(const vector<PT> &p) {
  double area = 0;
  for(int i = 0; i < p.size(); i++) {
    int j = (i + 1) % p.size();
    area += p[i].x * p[j].y - p[j].x * p[i].y;
  }
  return area / 2.0;
}

double ComputeArea(const vector<PT> &p) {
  return fabs(ComputeSignedArea(p));
}

PT ComputeCentroid(const vector<PT> &p) {
  PT c(0, 0);
  double scale = 6.0 * ComputeSignedArea(p);
  for (int i = 0; i < p.size(); i++){
    int j = (i + 1) % p.size();
    c = c + (p[i] + p[j]) * (p[i].x * p[j].y - p[j].x * p[i].y);
  }
  return c / scale;
}

// tests whether or not a given polygon (in CW or CCW order) is simple
bool IsSimple(const vector<PT> &p) {
  for (int i = 0; i < p.size(); i++) {
    for (int k = i + 1; k < p.size(); k++) {
      int j = (i + 1) % p.size();
      int l = (k + 1) % p.size();
      if (i == l || j == k) continue;
      if (SegmentsIntersect(p[i], p[j], p[k], p[l]))
        return false;
    }
  }
  return true;
}
```

Data structures

Binary indexed trees

```
// Fenwick tree - sommes sur intervalle
// Generalisation a d'autres operations inversibles
// Requetes de sommes/mises a jour : O(log n)
// Memoire : O(n)
struct FenwickTree {
  vector<ll> tree;
  // a = [0 .. n-1]
  FenwickTree(int n) {
    tree = vector<ll>(n);
  }
  // a[x] += v
  void add(int x, ll v) {
    for (; x < (int)tree.size(); x |= (x + 1))
      tree[x] += v;
  }
  // a[0] + ... + a[r]
  ll sum(int r) {
    ll ans = 0;
    for (; r >= 0; r = (r & (r + 1)) - 1)
      ans += tree[r];
    return ans;
  }
  // a[l] + ... + a[r]
  ll sum(int l, int r) {
    return sum(r) - sum(l - 1);
  }
};

// Fenwick Tree 2D classique - Sommes sur rectangles
// Temps : O(log^2 n) par requete
// Memoire : O(n^2)
struct FenwickTree2D {
  vector<vector<int>> tree;
  // a = [1 .. n][1 .. m]
  FenwickTree2D(int n, int m) {
    tree = vector<vector<int>>(n);
    for (int i = 0; i < n; ++i)
      tree[i] = vector<int>(m);
  }
  // a[x][y] += v
  void add(int x, int y, int v) {
    for (; x < (int)tree.size(); x |= (x + 1))
      for (int j = y; j < (int)tree[x].size(); j |= (j + 1))
        tree[x][j] += v;
  }
  // sum (0 <= i <= x; 0 <= j <= y) a[i][j]
  int sum(int x, int y) {
    int ans = 0;
    for (; x >= 0; x = (x & (x + 1)) - 1)
      for (int j = y; j >= 0; j = (j & (j + 1)) - 1)
        ans += tree[x][j];
    return ans;
  }
  // sum (x1 <= x <= x2; y1 <= y <= y2) a[x][y]
  int sum(int x1, int y1, int x2, int y2) {
    return sum(x2, y2) - sum(x1 - 1, y2) - sum(x2, y1 - 1) +
           sum(x1 - 1, y1 - 1);
  }
};

// Fenwick tree 2D compresse
// Necessite deux passes sur les requetes
// Temps : O(log^2 n) par requete
// Memoire : O(n log n)
// /!\ type
struct FenwickTree2DCompressed {
  vector<vector<int>> tree;
  vector<vector<int>> nodes;
  // a = [1 .. n][1 .. m]
  FenwickTree2DCompressed(int n) {
    tree = vector<vector<int>>(n);
    nodes = vector<vector<int>>(n);
  }
  // premiere passe : a appeler sur toutes les requetes prevues
  void init_add(int x, int y) {
    for (; x < (int)tree.size(); x |= (x + 1))
      nodes[x].push_back(y);
  }
  void init_sum(int x, int y) {
    for (; x >= 0; x = (x & (x + 1)) - 1)
      nodes[x].push_back(y);
  }
  void init_sum(int x1, int y1, int x2, int y2) {
    init_sum(x2, y2);
    init_sum(x1 - 1, y2);
    init_sum(x2, y1 - 1);
    init_sum(x1 - 1, y1 - 1);
  }
  // a appeler a la fin de la premiere passe
  void init() {
    for (int x = 0; x < (int)tree.size(); ++x) {
      sort(nodes[x].begin(), nodes[x].end());
      tree[x] = vector<int>(nodes[x].size());
    }
  }
  // a[x][y] += v
  void add(int x, int y, int v) {
    for (; x < (int)tree.size(); x |= (x + 1))
      for (int j = lower_bound(nodes[x].begin(), nodes[x].end(), y)
                - nodes[x].begin(); j < (int)nodes[x].size(); j |= (j + 1))
        tree[x][j] += v;
  }
  // sum (0 <= i <= x; 0 <= j <= y) a[i][j]
  int sum(int x, int y) {
    int ans = 0;
    for (; x >= 0; x = (x & (x + 1)) - 1)
      for (int j = lower_bound(nodes[x].begin(), nodes[x].end(), y)
                - nodes[x].begin(); j >= 0; j = (j & (j + 1)) - 1)
        ans += tree[x][j];
    return ans;
  }
  // sum (x1 <= x <= x2; y1 <= y <= y2) a[x][y]
  int sum(int x1, int y1, int x2, int y2) {
```

```
    return sum(x2, y2) - sum(x1 - 1, y2) - sum(x2, y1 - 1) +
           sum(x1 - 1, y1 - 1);
  }
};
```

LCA

```
// Plus petit ancetre commun
// Pretraitement : O(n log n)
// Requete : O(log n)
// Memoire : O(n)
struct LCA {
  vector<vector<int>> adj;
  vector<vector<int>> prev;
  int h;
  vector<int> tin, tout;
  int timer;

  void dfs(int u, int p = 0) {
    tin[u] = ++timer;
    prev[u][0] = p;
    for (int i = 1; i < h; ++i)
      prev[u][i] = prev[prev[u][i - 1]][i - 1];
    for (int v: adj[u])
      if (v != p)
        dfs(v, u);
    tout[u] = ++timer;
  }

  // adj : liste d'adjacence de l'arbre
  // peut contenir eventuellement des aretes vers un parent
  LCA(vector<vector<int>> adj, int root) {
    int n = (int)adj.size();
    this->adj = adj;
    for (h = 0; (1 << h) <= n; ++h)
      ;
    prev = vector<vector<int>>(n, vector<int>(h));
    timer = 0;
    tin = vector<int>(n);
    tout = vector<int>(n);
    dfs(root);
  }

  bool is_ancestor(int u, int v) {
    return tin[u] <= tin[v] && tout[v] <= tout[u];
  }

  int lca(int u, int v) {
    if (is_ancestor(u, v)) return u;
    if (is_ancestor(v, u)) return v;
    for (int i = h - 1; i >= 0; --i) {
      if (!is_ancestor(prev[u][i], v))
        u = prev[u][i];
    }
    return prev[u][0];
  }
};
```

HLD

```
// A adapter au probleme (par exemple avec des poids)
struct Edge {
  int u, v;

  Edge() {}
  Edge(int u, int v) : u(u), v(v) {}

  void read() {
    scanf("%d%d", &u, &v);
    --u, --v;
  }

  int next(int x) {
    return x == u ? v : u;
  }

  bool operator == (const Edge& e) const {
    return u == e.u && v == e.v;
  }
};

// HLD
struct HLD {
  vector<vector<Edge>> adj;
  vector<int> subtreesz;
  vector<int> nodechain;
  vector<int> nodepos;
  vector<int> head;
  vector<int> tail;
  vector<int> length;
  vector<Edge> prev;
  vector<Edge> next;
  int nb_chains;

  void compute_subtree(int u, int p = -1) {
    subtreesz[u]++;
    for (Edge e: adj[u]) {
      int v = e.next(u);
      if (v != p) {
        prev[v] = e;
        compute_subtree(v, u);
        subtreesz[u] += subtreesz[v];
      }
    }
  }

  void construct(int u, int p = -1) {
    nodechain[u] = nb_chains - 1;
    nodepos[u] = length[nb_chains - 1]++;
    if (nodepos[u] == 0)
      head[nb_chains - 1] = u;
    tail[nb_chains - 1] = u;
    if (adj[u].size() == 1 && adj[u][0].next(u) == p) return;
      Edge maxe = adj[u][0].next(u) == p ? adj[u][1] : adj[u][0];
    int maxv = maxe.next(u);
    for (Edge e: adj[u]) {
      int v = e.next(u);
      if (v != p && subtreesz[v] > subtreesz[maxv]) {
```

```
        maxv = v;
        maxe = e;
      }
    }
    next[u] = maxe;
    construct(maxv, u);
    for (Edge e: adj[u]) {
      int v = e.next(u);
      if (v != p && v != maxv) {
        nb_chains++;
        construct(v, u);
      }
    }
  }

  HLD(vector<vector<Edge>> adj, int root) {
    int n = (int)adj.size();
    this->adj = adj;
    subtreesz = vector<int>(n);
    prev = vector<Edge>(n);
    compute_subtree(root);
    nodechain = vector<int>(n);
    nodepos = vector<int>(n);
    tail = vector<int>(n);
    head = vector<int>(n);
    length = vector<int>(n);
    next = vector<Edge>(n);
    nb_chains = 1;
    construct(root);
  }
};
```

Link-cut

```
// Link cut trees
// Tire de https://github.com/jaehyunp/stanfordacm/
enum { LEFT, RIGHT };

struct node {
  node * parent, * child[2];
  int value, subtree_value;
  bool flip;

  void update(bool change = false, int new_value = 0) {
    if (change) value = new_value;
    subtree_value = value;
    FOR (d, 2)
      if (child[d] != nullptr)
        subtree_value += child[d]->subtree_value;
  }

  void propagate() {
    if (!flip) return;
    swap(child[LEFT], child[RIGHT]);
    FOR (d, 2) if (child[d] != nullptr) child[d]->flip ^= 1;
    flip = false;
  }

  void set_child(bool d, node * x) {
    if ((child[d] = x) != nullptr) x->parent = this;
    update();
  }

  bool has_child(node * x) { return child[LEFT] == x || child[RIGHT] == x; }
  bool is_root() { return parent == nullptr || !parent->has_child(this); }
  bool child_direction(node * x) { return (child[LEFT] == x) ? LEFT : RIGHT; }

  void rotate() {
    node *g = parent->parent;

    if (g != nullptr) g->propagate();
    parent->propagate();
    this->propagate();

    bool d = parent->child_direction(this);
    parent->set_child(d, child[!d]);
    this->set_child(!d, parent);
    if (g == nullptr || !g->has_child(parent)) parent = g;
    else g->set_child(g->child_direction(parent), this);
  }

  node * splay() {
    while (!is_root()) {
      node * p = parent;
      rotate();
      if (!is_root()) rotate();
      if (has_child(p->parent)) p->rotate();
    }
    return this;
  }

  node * leftmost_child() {
    for (node * x = this; ; x = x->child[LEFT]) {
      x->propagate();
      if (x->child[LEFT] == nullptr)
        return x->splay();
    }
  }
} nodes[MAXN];

node * expose(node * v) {
  node * old_preferred = nullptr;
  for (node * x = v; x != nullptr; x = x->parent) {
    x->splay()->propagate();
    x->set_child(RIGHT, old_preferred);
    old_preferred = x;
  }
  return v->splay();
}

void reroot(node * v) {
  expose(v)->flip ^= 1;
  v->parent = nullptr;
}

node * find_root(node * v) {
  return expose(v)->leftmost_child();
```

```
}

bool connected(node * u, node * v) {
  expose(u);
  return expose(v)->parent != nullptr;
}

bool link(node * u, node * v) {
  if (find_root(u) == find_root(v)) return false;
  reroot(v);
  v->parent = u;
  return true;
}

bool cut(node * u, node * v) {
  reroot(u);
  expose(v);
  if (v->child[RIGHT] != u || u->child[LEFT] != nullptr) return false;
  v->child[RIGHT]->parent = nullptr;
  v->child[RIGHT] = nullptr;
}

int query(node * u, node * v) {
  if (find_root(u) != find_root(v)) return -1;
  reroot(u);
  return expose(v)->subtree_value;
}
```

Convex hull trick

```
// Convex hull trick "statique"
// Recurrence de la forme : dp[i] = min(dp[j] + b[j] * a[i], j < i)
// Hypotheses : a[i] <= a[i+1] (requetes croissantes)
//              b[i] >= b[i+1] (coefficients directeurs decroissants)
const int MAX_NB_LINES = 100 * 1000;
struct ConvexHullTrick {
  ll a[MAX_NB_LINES], b[MAX_NB_LINES];
  ll hd, tl;

  ConvexHullTrick() {hd = tl = 0;}

  // ajoute y = a0 * x + b0
  void add(ll a0, ll b0) {
    while (tl - hd >= 2) {
      ll a1 = a[tl - 1], b1 = b[tl - 1];
      ll a2 = a[tl - 2], b2 = b[tl - 2];
      if ((long double)(b0 - b2) * (a1 - a0) <
          (long double)(b0 - b1) * (a2 - a0))
        break;
      //if ((long double)1 * (b0 - b2) / (a2 - a0) <
      //    (long double)1 * (b0 - b1) / (a1 - a0))
      //  break;
      tl--;
    }
    a[tl] = a0;
    b[tl++] = b0;
  }

  // valeur de l'enveloppe min sur la droite d'equation x = x0
  ll query(ll x0) {
    while (tl - hd >= 2) {
      if (x0 * a[hd] + b[hd] < x0 * a[hd + 1] + b[hd + 1])
        break;
      ++hd;
    }
    return x0 * a[hd] + b[hd];
  }
};

// Inspire de https://github.com/niklasb/contest-algos/
// Variante du precedent sans hypothese sur les a[i] et les b[i]
const ll QUERY = -(1LL << 62);
struct Line {
  ll a, b;
  mutable function<const Line *()> succ;

  bool operator < (const Line& other) const {
    if (other.b != QUERY)
      return a > other.a;
    const Line *s = succ();
    if (s == NULL) return false;
    return b - s->b > (s->a - a) * other.a;
  }
};

struct DynamicConvexHullTrick : public multiset<Line> {
  bool bad(iterator y) {
    auto z = next(y);
    if (y == begin()) {
      if (z == end()) return false;
      return y->a == z->a && y->b >= z->b;
    }
    auto x = prev(y);
    if (z == end())
      return y->a == x->a && y->b >= x->b;
    return (long double)(x->b - y->b) * (z->a - y->a) >=
           (long double)(y->b - z->b) * (y->a - x->a);
  }

  void add(ll a, ll b) {
    auto y = insert({a, b, nullptr});
    y->succ = [=] { return next(y) == end() ? 0 : &*next(y); };

    if (bad(y)) {
      erase(y);
      return;
    }

    while (next(y) != end() && bad(next(y)))
      erase(next(y));

    while (y != begin() && bad(prev(y)))
      erase(prev(y));
  }

  ll query(ll x) {
    auto l = *lower_bound((Line){x, QUERY, nullptr});
```

```
    return l.a * x + l.b;
  }
};
```

Segment tree

```
// Segment tree
template <typename T>
struct SegmentTree {
  function<void (vector<T>&, int, int, int)> push_up;
  function<int (vector<T>&, int, int, int, T, T)> combine_results;
  function<void (vector<T>&, int, int, int, T)> update_node;
  function<void (vector<T>&, int, int, int, T)> update_push;
  function<void (vector<T>&, vector<T>&, int, int, int)> push_down;
  vector<T> tree;
  vector<T> push_tree;
  T neutral;
  int n;

  SegmentTree(int n, T neutral = T(),
                     function<void (vector<T>&, int, int, int)>
                       push_up = nullptr,
                     function<int (vector<T>&, int, int, int, T, T)>
                       combine_results = nullptr,
                     function<void (vector<T>&, int, int, int, T)>
                       update_node = nullptr,
                     function<void (vector<T>&, int, int, int, T)>
                       update_push = nullptr,
                     function<void (vector<T>&, vector<T>&, int, int, int)>
                       push_down = nullptr) {
    this->neutral = neutral;
    this->push_up = push_up;
    this->combine_results = combine_results;
    this->update_node = update_node;
    this->update_push = update_push;
    this->push_down = push_down;
    int h;
    for (h = 0; (1 << h) < n; ++h)
      ;
    this->n = n = 1 << h;
    tree = vector<T>(2 * n, neutral);
    push_tree = vector<T>(2 * n, neutral);
  }

  void update(int ql, int qr, int l, int r, int u, T val) {
    if (ql <= l && r <= qr) {
      update_node(tree, u, l, r, val);
      if (update_push) update_push(push_tree, u, l, r, val);
      return;
    }
    if (push_down) push_down(tree, push_tree, u, l, r);
    int m = (l + r) / 2;
    if (ql <= m)
      update(ql, qr, l, m, 2 * u, val);
    if (qr > m)
      update(ql, qr, m + 1, r, 2 * u + 1, val);
    push_up(tree, u, l, r);
  }

  void update(int ql, int qr, T val) {
    update(ql, qr, 0, n - 1, 1, val);
  }

  T query(int ql, int qr, int l, int r, int u) {
    if (ql <= l && r <= qr)
      return tree[u];
    if (push_down) push_down(tree, push_tree, u, l, r);
    int m = (l + r) / 2;
    if (qr <= m)
      return combine_results(tree, u, l, r,
                             query(ql, qr, l, m, 2 * u), neutral);
    if (ql > m)
      return combine_results(tree, u, l, r, neutral,
                             query(ql, qr, m + 1, r, 2 * u + 1));
    return combine_results(tree, u, l, r,
                           query(ql, qr, l, m, 2 * u),
                           query(ql, qr, m + 1, r, 2 * u + 1));
  }

  T query(int ql, int qr) {
    return query(ql, qr, 0, n - 1, 1);
  }
};
```

Treap

```
// Treap
// Tire de https://github.com/jaehyunp/stanfordacm/
typedef int value;

enum { LEFT, RIGHT };
struct node {
  int size, priority;
  value x, subtree;
  node *child[2];
  node(const value &x): size(1), x(x), subtree(x) {
    priority = rand();
    child[0] = child[1] = nullptr;
  }
};

inline int size(const node *a) { return a == nullptr ? 0 : a->size; }

inline void update(node *a) {
  if (a == nullptr) return;
  a->size = size(a->child[0]) + size(a->child[1]) + 1;
  a->subtree = a->x;
  if (a->child[LEFT] != nullptr)
    a->subtree = a->child[LEFT]->subtree + a->subtree;
  if (a->child[RIGHT] != nullptr)
    a->subtree = a->subtree + a->child[RIGHT]->subtree;
}

node *rotate(node *a, bool d) {
  node *b = a->child[d];
  a->child[d] = b->child[!d];
  b->child[!d] = a;
  update(a); update(b);
```

```
  return b;
}

node *insert(node *a, int index, const value &x) {
  if (a == nullptr && index == 0) return new node(x);
  int middle = size(a->child[LEFT]);
  bool dir = index > middle;
  if (!dir) a->child[LEFT]  = insert(a->child[LEFT], index, x);
  else      a->child[RIGHT] = insert(a->child[RIGHT], index - middle - 1, x);
  update(a);
  if (a->priority > a->child[dir]->priority) a = rotate(a, dir);
  return a;
}

node *erase(node *a, int index) {
  assert(a != nullptr);
  int middle = size(a->child[LEFT]);
  if (index == middle) {
    if (a->child[LEFT] == nullptr && a->child[RIGHT] == nullptr) {
      delete a;
      return nullptr;
    } else if (a->child[LEFT] == nullptr) a = rotate(a, RIGHT);
    else if (a->child[RIGHT] == nullptr) a = rotate(a, LEFT);
    else a = rotate(a, a->child[LEFT]->priority < a->child[RIGHT]->priority);
    a = erase(a, index);
  } else {
    bool dir = index > middle;
    if (!dir) a->child[LEFT] = erase(a->child[LEFT], index);
    else      a->child[RIGHT] = erase(a->child[RIGHT], index - middle - 1);
  }
  update(a);
  return a;
}

void modify(node *a, int index, const value &x) {
  assert(a != nullptr);
  int middle = size(a->child[LEFT]);
  if (index == middle) a->x = x;
  else {
    bool dir = index > middle;
    if (!dir) modify(a->child[LEFT], index, x);
    else      modify(a->child[RIGHT], index - middle - 1, x);
  }
  update(a);
}

value query(node *a, int l, int r) {
  assert(a != nullptr);
  if (l <= 0 && size(a) - 1 <= r) return a->subtree;
  int middle = size(a->child[LEFT]);
  if (r < middle) return query(a->child[LEFT], l, r);
  if (middle < l) return query(a->child[RIGHT], l - middle - 1, r - middle - 1);
  value res = a->x;
  if (l < middle && a->child[LEFT] != nullptr)
    res = query(a->child[LEFT], l, r) + res;
  if (middle < r && a->child[RIGHT] != nullptr)
    res = res + query(a->child[RIGHT], l - middle - 1, r - middle - 1);
  return res;
}
```

Monotonic queue

```
// File monotone (min/max sur une fenetre glissante)
// Par defaut : file MIN
template<class T>
struct MonotonicQueue {
  deque<pair<T, int>> q;

  // ajoute elt a la file au "temps" t
  // t doit etre croissant au fur et a mesure des appels a add
  void add(T elt, int t) {
    while (!q.empty() && q.back().first > elt)
      q.pop_back();
    q.push_back({elt, t});
  }

  // supprime l'element issu du temps t
  // retourne vrai ssi. il y a effectivement un tel element
  // les appels successifs a remove doivent etre les memes qu'a add
  // (et dans le meme ordre)
  bool remove(int t) {
    if (!q.empty() && q.front().second == t) {
      q.pop_front();
      return true;
    }
    return false;
  }

  T get() {
    return q.front().first;
  }
};
```

Union-Find

```
// Union-Find
// Temps : O(alpha(n)) amorti par requete
// Memoire : O(n)
struct UnionFind {
  vector<int> cc;
  vector<int> ccsz;

  UnionFind() {}

  UnionFind(int n) {
    cc = vector<int>(n);
    ccsz = vector<int>(n);
    for (int i = 0; i < n; ++i)
      cc[i] = i;
  }

  int find(int i) {
    if (cc[i] != i)
      cc[i] = find(cc[i]);
    return cc[i];
  }
```

```
  bool merge(int i, int j) {
    i = find(i);
    j = find(j);
    if (i == j) return false;
    if (ccsz[i] < ccsz[j])
      swap(i, j);
    ccsz[i] += ccsz[j];
    cc[j] = i;
    return true;
  }
};
```

Strings

Aho-Corasick

```
// Tire de https://github.com/stjepang/snippets/blob/master/aho_corasick.cpp
// Aho Corasick
//
// Given a set of patterns, it builds the Aho-Corasick trie. This trie allows
// searching all matches in a string in linear time.
//
// To use, first call `node` once to create the root node, then call `insert`
// for every pattern, and finally initialize the trie by calling `init_aho`.
// Note: It is assumed all strings contains uppercase letters only.
//
// Globals:
// - V is the number of vertices in the trie
// - trie[x][c] is the child of node x labeled with letter 'A' + c
// - fn[x] points from node x to it's "failure" node
//
// Time complexity: O(N), where N is the sum of lengths of all patterns
//
// Constants to configure:
// - MAX is the maximum sum of lengths of patterns
// - ALPHA is the size of the alphabet (usually 26)
const int MAX = 1000;
const int ALPHA = 26;
int V;
int trie[MAX][ALPHA];
int fn[MAX];

int node() {
  for (int i = 0; i < ALPHA; ++i) trie[V][i] = 0;
  fn[V] = 0;
  return V++;
}

int insert(char *s) {
  int t = 0;
  for (; *s; ++s) {
    int c = *s - 'A';
    if (trie[t][c] == 0) trie[t][c] = node();
    t = trie[t][c];
  }
  return t;
}

void init_aho() {
  queue<int> Q;
  Q.push(0);

  while (!Q.empty()) {
    int t = Q.front(); Q.pop();

    for (int c = 0; c < ALPHA; ++c) {
      int x = trie[t][c];
      if (x) {
        Q.push(x);

        if (t) {
```

```
          fn[x] = fn[t];
          while (fn[x] && trie[fn[x]][c] == 0) fn[x] = fn[fn[x]];
          if (trie[fn[x]][c]) fn[x] = trie[fn[x]][c];
        }
      }
    }
  }
}
```

Palindromic tree

```
// Tire de https://github.com/stjepang/snippets/blob/master/palindromic_tree.cpp
// Palindromic tree
//
// Given a string, consider all its palindromic substrings.
// Denote every palindrome by its radius inside out. For example, denote the
// palindrome 'abcba' by 'cba'.
// This algorithm constructs a trie of all such radiuses.
//
// The algorithm is very similar to Aho-Corasick.
// More information:
// - http://adilet.org/blog/25-09-14/
// - http://codeforces.com/blog/entry/13959
//
// To run, set N and s, then call paltree().
// Note: It is assumed the string contains uppercase letters only.
//
// Globals:
// - N is the length of the string
// - s is the string
// - V is the number of vertices in the trie
// - trie[x][c] is the child of node x labeled with letter 'A' + c
// - fn[x] points from node x to it's "failure" node
// - len[x] is the depth of node x
//
// The root of even palindromes is node 0.
// The root of odd palindromes is node 1.
//
// Time complexity: O(N)
//
// Constants to configure:
// - MAX is the maximum length of the string
// - ALPHA is the size of the alphabet (usually 26)
const int MAX = 1000;
const int ALPHA = 26;
int N;
char s[MAX];

int V;
int trie[MAX][ALPHA];
int fn[MAX], len[MAX];

int node() {
  for (int i = 0; i < ALPHA; ++i) trie[V][i] = 0;
  fn[V] = len[V] = 0;
  return V++;
}

int suffix(int t, int i) {
  while (i - len[t] - 1 < 0 || s[i - len[t] - 1] != s[i])
    t = fn[t];
  return t;
}

void paltree() {
  V = 0; node(); node();
  len[0] =  0; fn[0] = 1;
  len[1] = -1; fn[1] = 0;

  int t = 0;
  for (int i = 0; i < N; ++i) {
    int c = s[i] - 'A';
    t = suffix(t, i);

    int &x = trie[t][c];
    if (!x) {
      x = node();
      len[x] = len[t] + 2;
      fn[x] = t == 1 ? 0 : trie[suffix(fn[t], i)][c];
    }
    t = x;
  }
}
```

Suffix array

```
// Suffix Array & LCP
// Construction en O(n log^2 n)
struct SuffixArray {
  // sa[i] = pos du premier caractere du i-eme suffixe dans l'ordre
  // lexicographique
  vector<int> sa;
  // pos[i] a l'issue du constructeur : position de s[i .. n-1] dans sa
  vector<int> pos;
  // lcp[i] : plus grand prefixe commun a sa[i] et sa[i+1]
  vector<int> lcp;
  int gap;
  int n;

  struct Compare {
    const SuffixArray& s;
    Compare(const SuffixArray& s) : s(s) {}

    bool operator () (int i, int j) const {
      if (s.pos[i] != s.pos[j]) return s.pos[i] < s.pos[j];
      i += s.gap;
      j += s.gap;
      return i < s.n && j < s.n ? s.pos[i] < s.pos[j] : i > j;
    }
  };

  SuffixArray(string s) {
    n = (int)s.length();
    sa = vector<int>(n);
    vector<int> tmp = vector<int>(n);
    pos = vector<int>(n);
    lcp = vector<int>(n - 1);
```

```
    // Construction de sa et pos
    for (int i = 0; i < n; ++i) {
      sa[i] = i;
      pos[i] = s[i];
    }

    for (gap = 1; gap <= n; gap *= 2) {
      sort(sa.begin(), sa.end(), Compare(*this));
      for (int i = 1; i < n; ++i)
        tmp[i] = tmp[i - 1] + Compare(*this)(sa[i - 1], sa[i]);
      for (int i = 0; i < n; ++i) pos[sa[i]] = tmp[i];
    }

    // Construction de lcp (peut etre supprime si non necessaire)
    int k = 0;
    for (int i = 0; i < n; ++i)
      if (pos[i] != n - 1) {
        int j = sa[pos[i] + 1];
        while (s[i + k] == s[j + k]) ++k;
        lcp[pos[i]] = k;
        if (k > 0) --k;
      }
  }
};
```

Z function

```
// Fonction Z : s[0 .. n-1] -> z[0 .. n-1]
// z[i] = longueur du plus grand prefixe commun de s[0 .. n-1] et s[i .. n-1]
// Calcul en O(n)
// Applications
// - Trouver toutes les occurrences d'un motif dans une chaine en O(n+p).
// - Nombre de sous-chaines distinctes quand on ajoute un caractere en tete.
// - Factorisation
vector<int> compute_z(string s) {
  int n = (int)s.length();
  vector<int> z(n);
  int l = 1, r = 0;
  for (int i = 1; i < n; ++i) {
    if (i <= r)
      z[i] = min(r - i + 1, z[i - 1]);
    while (i + z[i] < n && s[z[i]] == s[i + z[i]])
      ++z[i];
    if (i + z[i] - 1 > r) {
      l = i;
      r = i + z[i] - 1;
    }
  }
  z[0] = n;
  return z;
}
```

Prefix function

```
// Fonction prefixe : s[0 .. n-1] -> pref[0 .. n-1]
// pref[i] = max(0 <= k <= i | s[0 .. k-1] = s[i-k+1 .. i])
// O(n)
// Applications :
// - Trouver toutes les occurrences d'un motif (KMP)
// - Compter le nombre d'occurrences de chaque prefixe de s dans s
// - Nombre de sous-chaines distinctes dans s
// - Factorisation
vector<int> compute_prefix(string s) {
  int n = (int)s.length();
  vector<int> pref(n);
  for (int i = 1; i < n; ++i) {
    int j = pref[i - 1];
    while (j > 0 && s[i] != s[j])
      j = pref[j - 1];
    if (s[i] == s[j])
      ++j;
    pref[i] = j;
  }
  return pref;
}
```

Min rotation

```
// Tire de https://github.com/stjepang/snippets/blob/master/min_rotation.cpp
// Lexicographically minimum rotation of a sequence
//
// Given a sequence s of length N, min_rotation(s, N) returns the start index
// of the lexicographically minimum rotation.
//
// Note: array s must be of length of at least 2 * N.
//
// Time complexity: O(N)
int min_rotation(int *s, int N) {
  for (int i = 0; i < N; ++i)
    s[N + i] = s[i];

  int a = 0;
  for (int b = 0; b < N; ++b)
    for (int i = 0; i < N; ++i) {
      if (a + i == b || s[a + i] < s[b + i]) {
        b += max(0, i - 1);
        break;
      }
      if (s[a + i] > s[b + i]) {
        a = b;
        break;
      }
    }
  return a;
}
```

Manacher

```
// Tire de https://github.com/stjepang/snippets/blob/master/manacher.cpp
// Finds all palindromes in a string
//
// Given a string s of length N, finds all palindromes as its substrings.
//
// After calling manacher(s, N, rad), rad[x] will be the radius of the largest
// palindrome centered at index x / 2.
// Example:
//   s   = b a n a n a a
//   rad = 0000102010010
```

```
//
// Note: Array rad must be of length at least twice the length of the string.
// Also, "invalid" characters are denoted by -1, therefore the string must not
// contain such characters.
//
// Time complexity: O(N)
//
// Constants to configure:
// - MAX is the maximum length of the string
const int MAX = 1000;
void manacher(char *s, int N, int *rad) {
  static char t[2 * MAX];
  int m = 2 * N - 1;
  fill(t, t + m, -1);
  for (int i = 0; i < N; ++i) t[2 * i] = s[i];

  int x = 0;
  for (int i = 1; i < m; ++i) {
    int& r = rad[i] = 0;
    if (i <= x + rad[x])
      r = min(rad[x + x - i], x + rad[x] - i);
    while (i - r - 1 >= 0 && i + r + 1 < m && t[i - r - 1] == t[i + r + 1])
      ++r;
    if (i + r >= x + rad[x])
      x = i;
  }

  for (int i = 0; i < m; ++i)
    if (i - rad[i] == 0 || i + rad[i] == m - 1)
      ++rad[i];

  for (int i = 0; i < m; ++i)
    rad[i] /= 2;
}
```

Suffix automaton

```
// Automate suffixe
// Tire de https://github.com/ngthanhtrung23/ACM_Notebook_new/
struct Node {
  int len, link; // len = max length of suffix in this class
  int next[33];
};
Node s[MN * 2];
set<pair<int,int>> order; // in most application we'll need to sort by len

struct Automaton {
  int sz, last;

  Automaton() {
    order.clear();
    sz = last = 0;
    s[0].len = 0;
    s[0].link = -1;
    ++sz;
    // need to reset next if necessary
  }

  void extend(char c) {
    c = c - 'A';
    int cur = sz++, p;
    s[cur].len = s[last].len + 1;
    order.insert(make_pair(s[cur].len, cur));

    for (p = last; p != -1 && !s[p].next[c]; p = s[p].link)
      s[p].next[c] = cur;
    if (p == -1) s[cur].link = 0;
    else {
      int q = s[p].next[c];
      if (s[p].len + 1 == s[q].len) s[cur].link = q;
      else {
        int clone = sz++;
        s[clone].len = s[p].len + 1;
        memcpy(s[clone].next, s[q].next, sizeof s[q].next);
        s[clone].link = s[q].link;
        order.insert(make_pair(s[clone].len, clone));

        for (; p != -1 && s[p].next[c] == q; p = s[p].link)
          s[p].next[c] = clone;
        s[q].link = s[cur].link = clone;
      }
    }
    last = cur;
  }
};

// Construct:
// Automaton sa; for(char c : s) sa.extend(c);
// 1. Number of distinct substr:
//    - Find number of different paths --> DFS on SA
//    - f[u] = 1 + sum( f[v] for v in s[u].next
// 2. Number of occurrences of a substr:
//    - Initially, in extend: s[cur].cnt = 1; s[clone].cnt = 0;
//    - for(it : reverse order)
//        p = nodes[it->second].link;
//        nodes[p].cnt += nodes[it->second].cnt
// 3. Find total length of different substrings:
//    - We have f[u] = number of strings starting from node u
//    - ans[u] = sum(ans[v] + d[v] for v in next[u])
// 4. Lexicographically k-th substring
//    - Based on number of different substring
// 5. Smallest cyclic shift
//    - Build SA of S+S, then just follow smallest link
// 6. Find first occurrence
//    - firstpos[cur] = len[cur] - 1, firstpos[clone] = firstpos[q]
```

String hash

```
// Hash
// P premier de l'ordre de la taille de l'alphabet
// P = 31 pour les minuscules, 53 pour tous les caracteres alphabetiques
//
// (Quelques) applications :
// - Recherche de motifs (Rabin-Karp)
// - Nombre de sous-chaines distinctes
// - Requetes : est-ce que la sous-chaine est un palindrome ?
//
// /!\ Ne pas utiliser les hash modulo 2^64
```

```cpp
// http://codeforces.com/blog/entry/4898(
//
// Utiliser un modulo suffit generalement (/!\ paradoxe des anniversaires)
// Sinon utiliser deux mod
// Quelques premiers de l'ordre de 10^9 : 10^9 + 7 ; 10^9 + 9 ; 10^9 + 123
// Trouver le reste entre 0 et M - 1 de a/M : (a % M + M) % M
typedef long long ll;
ll posmod(ll a, ll M) { return (a % M + M) % M; }

struct StringHash {
  ll mod;
  vector<ll> ppows;
  // hashsuff[i] = s[i] + s[i+1] * P + s[i+2] * P^2 + ... + s[n-1] * P^(n-1-i)
  //               [mod]
  vector<ll> hashsuff;

  StringHash() {}

  StringHash(string s, ll base, ll mod) {
    this->mod = mod;
    int n = (int)s.length();
    ppows = vector<ll>(n);
    hashsuff = vector<ll>(n + 1);
    ppows[0] = 1;
    for (int i = 1; i < n; ++i)
      ppows[i] = posmod(ppows[i - 1] * base, mod);
    hashsuff[n] = 0;
    for (int i = n - 1; i >= 0; --i)
      hashsuff[i] = posmod(hashsuff[i + 1] * base + s[i], mod);
  }

  // [l, r]
  // 0 <= l <= r <= n
  ll get_hash(int l, int r) {
    return posmod(hashsuff[l] - hashsuff[r + 1] * ppows[r + 1 - l], mod);
  }
};

const int BASE = 31;
const int NB_MODS = 2;
const ll MOD[NB_MODS] = {1000 * 1000 * 1000 + 9, 1000 * 1000 * 1000 + 7};

// Premiers :
//
// 2 3 5 7 11 13 17 19 23 29
// 31 37 41 43 47 53 59 61 67 71
// 73 79 83 89 97 101 103 107 109 113
// 127 131 137 139 149 151
//
// ...
//
// 1009 1013 1019 1021 1031 1033 1039 1049 1051 1061
//
// ...
//
// 1000003 1000033 1000037 1000039
// 1000081 1000099 1000117 1000121
// 1000133 1000151 1000159 1000171
// 1000183 1000187 1000193 1000199
// 1000211 1000213 1000231 1000249
//
// ...
//
// 1000000007 1000000009 1000000021 1000000033
// 1000000087 1000000093 1000000097 1000000103
// 1000000123 1000000181 1000000207 1000000223
// 1000000241 1000000271 1000000289 1000000297
// 1000000321 1000000349 1000000363 1000000403
```

Formulas

Bernoulli numbers:

n	0	1	2	4	6	8	10	12
B_n	1	-1/2	1/6	-1/30	1/42	-1/30	5/66	-691/2730

$B_{2n+3} = 0$

$(m+1)B_m = -\sum_{k=0}^{m-1} \binom{k}{m+1} B_k$

$\sum_{k=1}^{n} k^m = \frac{1}{m+1} \sum_{k=0}^{m} \binom{m+1}{k} B_k n^{m+1-k}$.

$\sum_{k=a}^{b-1} f(k) = \int_a^b f(t)dt + \sum_{k=1}^{n} \frac{B_k}{k!} \left(f^{(k-1)}(b) - f^{(k-1)}(a)\right) + \frac{(-1)^{n+1}}{n!} \int_a^b B_n(t) f^{(n)}(t)dt.$

$n \geq 1$. $\zeta(2n) = (-1)^{n-1} 2^{2n-1} \frac{B_{2n}}{(2n)!} \pi^{2n}$.

Pick's theorem: $A = I + \frac{1}{2}B - 1$. (A area polygon, I number of interior points, B number of boundary points).

Triangle geometry

$p = \frac{a+b+c}{2}$.

Area: $A = \frac{\sin\alpha}{a} = \sqrt{p(p-a)(p-b)(p-c)}$.

Incircle: $r = \frac{A}{p}$. $I = \frac{a}{2p}A + \frac{b}{2p}B + \frac{c}{2p}C$.

Length of the angle bissector through A: $i_a = \frac{2}{b+c}\sqrt{p(p-a)bc}$.

Circumcircle: $R = \frac{abc}{4A} = \frac{a}{2\sin\alpha}$. $O = \frac{1}{\sin(2\alpha)+\sin(2\beta)+\sin(2\gamma)}(\sin(2\alpha)A + \sin(2\beta)B + \sin(2\gamma)C)$.

Orthocenter: $H = \frac{1}{\tan\alpha+\tan\beta+\tan\gamma}(\tan(\alpha)A + \tan(\beta)B + \tan(\gamma)C)$.

Law of cosines: $c^2 = a^2 + b^2 - 2ab\cos(\gamma)$.