University of Tartu ICPC Team Notebook (2017-2018)

October 30, 2017

Contents

1 gcc ordered set

```
#include <ext/pb_ds/assoc_container.hpp>
#include <ext/pb_ds/tree_policy.hpp>
using namespace __gnu_pbds;
template <typename T>
using ordered_set = tree<T, null_type, less<T>, rb_tree_tag, tree_order_statistics_node_update>;
int main(){
  ordered_set<int>  cur;
  cur.insert(1);
  cur.insert(3);
  cout << cur.order_of_key(2) << endl; // the number of elements in the set less than 2
  cout << *cur.find_by_order(0) << endl; // the 0-th smallest number in the set(0-based)
  cout << *cur.find_by_order(1) << endl; // the 1-th smallest number in the set(0-based)
}
```

2 Triangle centers

```
const double min_delta = 1e-13;
const double coord_max = 1e6;
typedef complex < double > point;
point A, B, C; // vertixes of the triangle
bool collinear(){
  double min_diff = min(abs(A - B), min(abs(A - C), abs(B - C)));
  if(min_diff < coord_max * min_delta)
    return true;
  point sp = (B - A) / (C - A);
  double ang = M_PI/2-abs(abs(arg(sp))-M_PI/2); //positive angle with the real line
  return ang < min_delta;
}
point circum_center(){
  if(collinear())
    return point(NAN,NAN);
  //squared lengths of sides
```

```
  double a2, b2, c2;
  a2 = norm(B - C);
  b2 = norm(A - C);
  c2 = norm(A - B);
  //barycentric coordinates of the circumcenter
  double c_A, c_B, c_C;
  c_A = a2 * (b2 + c2 - a2);//sin(2 * alpha) may be used as well
  c_B = b2 * (a2 + c2 - b2);
  c_C = c2 * (a2 + b2 - c2);
  double sum = c_A + c_B + c_C;
  c_A /= sum;
  c_B /= sum;
  c_C /= sum;
  // cartesian coordinates of the circumcenter
  return c_A * A + c_B * B + c_C * C;
}
point centroid(){ //center of mass
  return (A + B + C) / 3.0;
}
point ortho_center(){ //euler line
  point O = circum_center();
  return O + 3.0 * (centroid() - O);
};
point nine_point_circle_center(){ //euler line
  point O = circum_center();
  return O + 1.5 * (centroid() - O);
};
point in_center(){
  if(collinear())
    return point(NAN,NAN);
  double a, b, c; //side lengths
  a = abs(B - C);
  b = abs(A - C);
  c = abs(A - B);
  //trilinear coordinates are (1,1,1)
  //barycentric coordinates
  double c_A = a, c_B = b, c_C = c;
  double sum = c_A + c_B + c_C;
  c_A /= sum;
  c_B /= sum;
  c_C /= sum;
  // cartesian coordinates of the incenter
  return c_A * A + c_B * B + c_C * C;
}
```

3 2D line segment

```
const long double PI = acos(-1.0L);

struct Vec {
  long double x, y;

  Vec& operator-=(Vec r) {
    x -= r.x, y -= r.y;
    return *this;
  }
  Vec operator-(Vec r) {return Vec(*this) -= r;}

  Vec& operator+=(Vec r) {
    x += r.x, y += r.y;
    return *this;
  }
  Vec operator+(Vec r) {return Vec(*this) += r;}
  Vec operator-() {return {-x, -y};}
  Vec& operator*=(long double r) {
    x *= r, y *= r;
    return *this;
  }
  Vec operator*(long double r) {return Vec(*this) *= r;}
  Vec& operator/=(long double r) {
    x /= r, y /= r;
    return *this;
  }
```

```
  Vec operator/(long double r) {return Vec(*this) /= r;}

  long double operator*(Vec r) {
    return x * r.x + y * r.y;
  }
};
ostream& operator<<(ostream& l, Vec r) {
  return l << '(' << r.x << ", " << r.y << ')';
}
long double len(Vec a) {
  return hypot(a.x, a.y);
}
long double cross(Vec l, Vec r) {
  return l.x * r.y - l.y * r.x;
}
long double angle(Vec a) {
  return fmod(atan2(a.y, a.x)+2*PI, 2*PI);
}
Vec normal(Vec a) {
  return Vec({-a.y, a.x}) / len(a);
}
```

```
struct Segment {
  Vec a, b;
  Vec d() {
    return b-a;
  }
};
ostream& operator<<(ostream& l, Segment r) {
  return l << r.a << '-' << r.b;
}

Vec intersection(Segment l, Segment r) {
  Vec dl = l.d(), dr = r.d();
  if(cross(dl, dr) == 0)
    return {nanl(""), nanl("")};

  long double h = cross(dr, l.a-r.a) / len(dr);
  long double dh = cross(dr, dl) / len(dr);

  return l.a + dl * (h / -dh);
}

//Returns the area bounded by halfplanes
long double getArea(vector&lt;Segment&gt; lines) {
  long double lowerbound = -HUGE_VALL, upperbound = HUGE_VALL;

  vector&lt;Segment&gt; linesBySide[2];
  for(auto line : lines) {
    if(line.b.y == line.a.y) {
      if(line.a.x < line.b.x)
        lowerbound = max(lowerbound, line.a.y);
      else
        upperbound = min(upperbound, line.a.y);
    }
    else if(line.a.y < line.b.y)
      linesBySide[1].push_back(line);
    else
      linesBySide[0].push_back({line.b, line.a});
  }

  sort(linesBySide[0].begin(), linesBySide[0].end(), [](Segment l, Segment r) {
    if(cross(l.d(), r.d()) == 0) return normal(l.d())*l.a > normal(r.d())*r.a;
    return cross(l.d(), r.d()) < 0;
  });
  sort(linesBySide[1].begin(), linesBySide[1].end(), [](Segment l, Segment r) {
    if(cross(l.d(), r.d()) == 0) return normal(l.d())*l.a < normal(r.d())*r.a;
    return cross(l.d(), r.d()) > 0;
  });

  //Now find the application area of the lines and clean up redundant ones
  vector<long double> applyStart[2];
```

```
  for(int side = 0; side < 2; side++) {
    vector<long double> &apply = applyStart[side];
    vector&lt;Segment&gt; curLines;

    for(auto line : linesBySide[side]) {
      while(curLines.size() > 0) {
        Segment other = curLines.back();

        if(cross(line.d(), other.d()) != 0) {
          long double start = intersection(line, other).y;
          if(start > apply.back())
            break;
        }

        curLines.pop_back();
        apply.pop_back();
      }

      if(curLines.size() == 0)
        apply.push_back(-HUGE_VALL);
      else
        apply.push_back(intersection(line, curLines.back()).y);
      curLines.push_back(line);
    }

    linesBySide[side] = curLines;
  }
  applyStart[0].push_back(HUGE_VALL);
  applyStart[1].push_back(HUGE_VALL);

  long double result = 0;
  {
    long double lb = -HUGE_VALL, ub;
    for(int i=0, j=0; i < (int)linesBySide[0].size() && j < (int)linesBySide[1].size();lb = ub) {
      ub = min(applyStart[0][i+1], applyStart[1][j+1]);

      long double alb = lb, aub = ub;
      Segment l0 = linesBySide[0][i], l1 = linesBySide[1][j];

      if(cross(l1.d(), l0.d()) > 0)
        alb = max(alb, intersection(l0, l1).y);
      else if(cross(l1.d(), l0.d()) < 0)
        aub = min(aub, intersection(l0, l1).y);
      alb = max(alb, lowerbound);
      aub = min(aub, upperbound);
      aub = max(aub, alb);

      {
        long double x1 = l0.a.x + (alb - l0.a.y) / l0.d().y * l0.d().x;
        long double x2 = l0.a.x + (aub - l0.a.y) / l0.d().y * l0.d().x;
        result -= (aub - alb) * (x1 + x2) / 2;
      }
      {
        long double x1 = l1.a.x + (alb - l1.a.y) / l1.d().y * l1.d().x;
        long double x2 = l1.a.x + (aub - l1.a.y) / l1.d().y * l1.d().x;
        result += (aub - alb) * (x1 + x2) / 2;
      }

      if(applyStart[0][i+1] < applyStart[1][j+1])
        i++;
      else
        j++;
    }
  }
  return result;
}
```

4 Dinic

```
struct MaxFlow{
    typedef long long ll;
    const ll INF = 1e18;
    struct Edge{
```

```
        int u,v;
        ll c,rc;
        shared_ptr<ll> flow;
        pair<int,int> id() const {
            return make_pair(min(u,v),max(u,v));
        }
        Edge(int _u, int _v, ll _c, ll _rc = 0):u(_u),v(_v),c(_c),rc(_rc){
        }
        void join(const Edge &t){
            if(u == t.u){
                c += t.c;
                rc += t.rc;
            }
            else{
                c += t.rc;
                rc += t.c;
            }
        }
    };
    struct FlowTracker{
        shared_ptr<ll> flow;
        ll cap, rcap;
        bool dir;
        FlowTracker(ll _cap, ll _rcap, shared_ptr<ll> _flow, int
         ↪  _dir):cap(_cap),rcap(_rcap),flow(_flow),dir(_dir){ }
        ll rem() const {
            if(dir == 0){
                return cap-*flow;
            }
            else{
                return rcap+*flow;
            }
        }
        void add_flow(ll f){
            if(dir == 0)
                *flow += f;
            else
                *flow -= f;
            assert(*flow <= cap);
            assert(-*flow <= rcap);
        }
        operator ll() const { return rem(); }
        void operator-=(ll x){ add_flow(x); }
        void operator+=(ll x){ add_flow(-x); }
    };
    int source,sink;
    vector<vector<int> > adj;
    vector<vector<FlowTracker> > cap;
    vector<Edge> edges;
    MaxFlow(int _source, int _sink):source(_source),sink(_sink){
        assert(source != sink);
    }
    int add_edge(Edge e){
        edges.push_back(e);
        return edges.size()-1;

    }
    int add_edge(int u, int v, ll c, ll rc = 0){
        return add_edge(Edge(u,v,c,rc));
    }
    void group_edges(){
        map<pair<int,int>,vector<Edge> > edge_groups;
        for(auto edge: edges)
            if(edge.u != edge.v)
                edge_groups[edge.id()].push_back(edge);
        vector<Edge> grouped_edges;
        for(auto group: edge_groups){
            Edge main_edge = group.second[0];
            for(int i = 1; i < group.second.size(); ++i)
                main_edge.join(group.second[i]);
            grouped_edges.push_back(main_edge);
        }
        edges = grouped_edges;
```

```
    }
    vector<int> now,lvl;
    void prep(){
        int max_id = max(source,sink);
        for(auto edge : edges)
            max_id = max(max_id,max(edge.u,edge.v));
        adj.resize(max_id+1);
        cap.resize(max_id+1);
        now.resize(max_id+1);
        lvl.resize(max_id+1);
        for(auto &edge : edges){
            auto flow = make_shared<ll>(0);
            adj[edge.u].push_back(edge.v);
            cap[edge.u].push_back(FlowTracker(edge.c,edge.rc,flow,0));
            adj[edge.v].push_back(edge.u);
            cap[edge.v].push_back(FlowTracker(edge.c,edge.rc,flow,1));
            assert(cap[edge.u].back() == edge.c);
            edge.flow = flow;
        }
    }
    bool dinic_bfs(){
        fill(now.begin(),now.end(),0);
        fill(lvl.begin(),lvl.end(),0);
        lvl[source] = 1;
        vector<int> bfs(1,source);
        for(int i = 0; i < bfs.size(); ++i){
            int u = bfs[i];
            for(int j = 0; j < adj[u].size(); ++j){
                int v = adj[u][j];
                if(cap[u][j] > 0 && lvl[v] == 0){
                    lvl[v] = lvl[u]+1;
                    bfs.push_back(v);
                }
            }
        }
        return lvl[sink] > 0;
    }
    ll dinic_dfs(int u, ll flow){
        if(u == sink)
            return flow;
        while(now[u] < adj[u].size()){
            int v = adj[u][now[u]];
            if(lvl[v] == lvl[u] + 1 && cap[u][now[u]] != 0){
                ll res = dinic_dfs(v,min(flow,(ll)cap[u][now[u]]));
                if(res > 0){
                    cap[u][now[u]] -= res;
                    return res;
                }
            }
            ++now[u];
        }
        return 0;
    }
    ll calc(){
        prep();
        ll ans = 0;
        while(dinic_bfs()){
            ll cur = 0;
            do{
                cur = dinic_dfs(source,INF);
                ans += cur;
            }while(cur > 0);
        }
        return ans;
    }
};
int main(){
    int n,m;
    cin >> n >> m;
    auto mf = MaxFlow(1,n); // arguments source and sink, memory usage O(largest node index), sink doesn't
    ↪ need to be last
    int edge_index;
    for(int i = 0; i < m; ++i){
```

```
        int a,b,c;
        cin >> a >> b >> c;
        //undirected edge is a pair of edges (a,b,c,0) and (a,b,0,c)
        edge_index = mf.add_edge(a,b,c,c); //store edge index if care about flow value
    }
    mf.group_edges(); // small auxillary to remove multiple edges, only use this if we need to know TOTAL
    ↪ FLOW ONLY
    cout << mf.calc() << '\n';
    //cout << *mf.edges[edge_index].flow << '\n'; // ONLY if group_edges() WAS NOT CALLED
}
```

5 Min Cost Max Flow with succesive dijkstra $\mathcal{O}(\text{flow} \cdot n^2)$

```
const int nmax=1055;
const ll inf=1e14;
int t, n, v; //0 is source, v-1 sink
ll rem_flow[nmax][nmax]; //set [x][y] for directed capacity from x to y.
ll cost[nmax][nmax]; //set [x][y] for directed cost from x to y. SET TO inf IF NOT USED
ll min_dist[nmax];
int prev_node[nmax];
ll node_flow[nmax];
bool visited[nmax];
ll tot_cost, tot_flow; //output
void min_cost_max_flow(){ //incase of negative edges have to add Bellman-Ford that is run once.
  tot_cost=0;             //Does not work with negative cycles.
  tot_flow=0;
  ll sink_pot=0;
  while(true){
    for(int i=0; i<=v; ++i){
      min_dist[i]=inf;
      visited[i]=false;
    }
    min_dist[0]=0;
    node_flow[0]=inf;
    int min_node;
    while(true){ //Use Dijkstra to calculate potentials
      int min_node=v;
      for(int i=0; i<v; ++i){
        if((!visited[i]) && min_dist[i]<min_dist[min_node]){
          min_node=i;
        }
      }
      if(min_node==v){
        break;
      }
      visited[min_node]=true;
      for(int i=0; i<v; ++i){
        if((!visited[i]) && min_dist[min_node]+cost[min_node][i] < min_dist[i]){
          min_dist[i]=min_dist[min_node]+cost[min_node][i];
          prev_node[i]=min_node;
          node_flow[i]=min(node_flow[min_node], rem_flow[min_node][i]);
        }
      }
    }
    if(min_dist[v-1]==inf){
      break;
    }
    for(int i=0; i<v; ++i){   //Apply potentials to edge costs.
      for(int j=0; j<v; ++j){ //Found path from source to sink becomes 0 cost.
        if(cost[i][j]!=inf){
          cost[i][j]+=min_dist[i];
          cost[i][j]-=min_dist[j];
        }
      }
    }
    sink_pot+=min_dist[v-1];
    tot_flow+=node_flow[v-1];
    tot_cost+=sink_pot*node_flow[v-1];
    int cur=v-1;
    while(cur!=0){ //Backtrack along found path that now has 0 cost.
      rem_flow[prev_node[cur]][cur]-=node_flow[v-1];
      rem_flow[cur][prev_node[cur]]+=node_flow[v-1];
      cost[cur][prev_node[cur]]=0;
```

```
      if(rem_flow[prev_node[cur]][cur]==0){
        cost[prev_node[cur]][cur]=inf;
      }
      cur=prev_node[cur];
    }
  }
}
```

6 Min Cost Max Flow with Cycle Cancelling $\mathcal{O}(\text{flow} \cdot nm)$

```
struct Network {
  struct Node;

  struct Edge {
    Node *u, *v;
    int f, c, cost;

    Node* from(Node* pos) {
      if(pos == u)
        return v;
      return u;
    }
    int getCap(Node* pos) {
      if(pos == u)
        return c-f;
      return f;
    }
    int addFlow(Node* pos, int toAdd) {
      if(pos == u) {
        f += toAdd;
        return toAdd * cost;
      }
      else {
        f -= toAdd;
        return -toAdd * cost;
      }
    }

  };
  struct Node {
    vector<Edge*> conn;
    int index;
  };

  deque<Node> nodes;
  deque<Edge> edges;

  Node* addNode() {
    nodes.push_back(Node());
    nodes.back().index = nodes.size()-1;
    return &nodes.back();
  }
  Edge* addEdge(Node* u, Node* v, int f, int c, int cost) {
    edges.push_back({u, v, f, c, cost});
    u->conn.push_back(&edges.back());
    v->conn.push_back(&edges.back());
    return &edges.back();
  }


  //Assumes all needed flow has already been added
  int minCostMaxFlow() {
    int n = nodes.size();
    int result = 0;

    struct State {
      int p;
      Edge* used;
    };

    while(1) {
      vector<vector<State> > state(1, vector<State>(n, {0, 0}));

```

```
    for(int lev = 0; lev < n; lev++) {
      state.push_back(state[lev]);
      for(int i=0;i<n;i++)
        if(lev == 0 || state[lev][i].p < state[lev-1][i].p) {

          for(Edge* edge : nodes[i].conn) if(edge->getCap(&nodes[i]) > 0) {
            int np = state[lev][i].p + (edge->u == &nodes[i] ? edge->cost : -edge->cost);
            int ni = edge->from(&nodes[i])->index;

            if(np < state[lev+1][ni].p) {
              state[lev+1][ni].p = np;
              state[lev+1][ni].used = edge;
            }
          }
        }
    }

    //Now look at the last level
    bool valid = false;

    for(int i=0;i<n;i++)
      if(state[n-1][i].p > state[n][i].p) {
        valid = true;

        vector<Edge*> path;

        int cap = 1000000000;
        Node* cur = &nodes[i];
        int clev = n;

        vector<bool> explr(n, false);

        while(!explr[cur->index]) {
          explr[cur->index] = true;

          State cstate = state[clev][cur->index];
          cur = cstate.used->from(cur);

          path.push_back(cstate.used);
        }

        reverse(path.begin(), path.end() );

        {
          int i=0;
          Node* cur2 = cur;

          do {
            cur2 = path[i]->from(cur2);
            i++;
          }while(cur2 != cur);

          path.resize(i);
        }

        for(auto edge : path) {
          cap = min(cap, edge->getCap(cur));
          cur = edge->from(cur);
        }

        for(auto edge : path) {
          result += edge->addFlow(cur, cap);
          cur = edge->from(cur);
        }
      }

    if(!valid) break;
  }

  return result;
}

};
```

7 Aho Corasick $\mathcal{O}(|\text{alpha}| \sum \text{len})$

```
const int alpha_size=26;
struct node{
  node *nxt[alpha_size]; //May use other structures to move in trie
  node *suffix;
  node(){
    memset(nxt, 0, alpha_size*sizeof(node *));
  }
  int cnt=0;
};
node *aho_corasick(vector<vector<char> > &dict){
  node *root= new node;
  root->suffix = 0;
  vector<pair<vector<char> *, node *> > cur_state;
  for(vector<char> &s : dict)
    cur_state.emplace_back(&s, root);
  for(int i=0; !cur_state.empty(); ++i){
    vector<pair<vector<char> *, node *> > nxt_state;
    for(auto &cur : cur_state){
      node *nxt=cur.second->nxt[(*cur.first)[i]];
      if(nxt){
        cur.second=nxt;
      }else{
        nxt = new node;
        cur.second->nxt[(*cur.first)[i]] = nxt;
        node *suf = cur.second->suffix;
        cur.second = nxt;
        nxt->suffix = root; //set correct suffix link
        while(suf){
          if(suf->nxt[(*cur.first)[i]]){
            nxt->suffix = suf->nxt[(*cur.first)[i]];
            break;
          }
          suf=suf->suffix;
        }
      }
      if(cur.first->size() > i+1)
        nxt_state.push_back(cur);
    }
    cur_state=nxt_state;
  }
  return root;
}
//auxilary functions for searhing and counting
node *walk(node *cur, char c){ //longest prefix in dict that is suffix of walked string.
  while(true){
    if(cur->nxt[c])
      return cur->nxt[c];
    if(!cur->suffix){
      return cur;
    }
    cur = cur->suffix;
  }
}
void cnt_matches(node *root, vector<char> &match_in){
  node *cur = root;
  for(char c : match_in){
    cur = walk(cur, c);
    ++cur->cnt;
  }
}
void add_cnt(node *root){ //After counting matches propagete ONCE to suffixes for final counts
  vector<node *> to_visit = {root};
  for(int i=0; i<to_visit.size(); ++i){
    node *cur = to_visit[i];
    for(int j=0; j<alpha_size; ++j){
      if(cur->nxt[j]){
        to_visit.push_back(cur->nxt[j]);
      }
    }
  }
  for(int i=to_visit.size()-1; i>0; --i){
```

```
    to_visit[i]->suffix->cnt += to_visit[i]->cnt;
  }
}
```

8 Suffix automaton $O((n+q)\log(|\textbf{alpha}|))$

```
class AutoNode {
 private:
  map< char, AutoNode * > nxt_char;  // Map is faster than hashtable and unsorted arrays
 public:
  int len; //Length of longest suffix in equivalence class.
  AutoNode *suf;
  bool has_nxt(char c) const {
    return nxt_char.count(c);
  }
  AutoNode *nxt(char c) {
    if (!has_nxt(c))
      return NULL;
    return nxt_char[c];
  }
  void set_nxt(char c, AutoNode *node) {
    nxt_char[c] = node;
  }
  AutoNode *split(int new_len, char c) {
    AutoNode *new_n = new AutoNode;
    new_n->nxt_char = nxt_char;
    new_n->len = new_len;
    new_n->suf = suf;
    suf = new_n;
    return new_n;
  }
  // Extra functions for matching and counting
  AutoNode *lower_depth(int depth) { //move to longest suffix of current with a maximum length of depth.
    if (suf->len >= depth)
      return suf->lower_depth(depth);
    return this;
  }
  AutoNode *walk(char c, int depth, int &match_len) { //move to longest suffix of walked path that is a substring
    match_len = min(match_len, len);                  //includes depth limit(needed for finding matches)
    if (has_nxt(c)) {                                  //as suffixes are in classes match_len must be tracte eternally
      ++match_len;
      return nxt(c)->lower_depth(depth);
    }
    if (suf)
      return suf->walk(c, depth, match_len);
    return this;
  }
  int paths_to_end = 0;
  void set_as_end() { //All suffixes of current node are marked as ending nodes.
    paths_to_end = 1;
    if (suf) suf->set_as_end();
  }
  bool vis = false;
  void calc_paths_to_end() {   //Call ONCE from ROOT. For each node calculates number of ways to reach an end node.
    if (!vis) {                //paths_to_end is ocurence count for any strings in current suffix equivalence class.
      vis = true;
      for (auto cur : nxt_char) {
        cur.second->calc_paths_to_end();
        paths_to_end += cur.second->paths_to_end;
      }
    }
  }
};
struct SufAutomaton {
  AutoNode *last;
  AutoNode *root;
  void extend(char new_c) {
    AutoNode *new_end = new AutoNode;  // The equivalence class containing the whole new string
    new_end->len = last->len + 1;
```

```
    AutoNode *suf_w_nxt = last;                          // The whole old string class
    while (suf_w_nxt && !suf_w_nxt->has_nxt(new_c)) {   // is turned into the longest suffix which
                                                         // can be turned into a substring of old state
                                                         // by appending new_c
      suf_w_nxt->set_nxt(new_c, new_end);
      suf_w_nxt = suf_w_nxt->suf;
    }
    if (!suf_w_nxt) {  // The new character isn't part of the old string
      new_end->suf = root;
    } else {
      AutoNode *max_sbstr = suf_w_nxt->nxt(new_c);  // Equivalence class containing longest
                                                     // substring which is a suffix of the new state.
      if (suf_w_nxt->len + 1 == max_sbstr->len) { // Check whether splitting is needed
        new_end->suf = max_sbstr;
      } else {
        AutoNode *eq_sbstr = max_sbstr->split(suf_w_nxt->len + 1, new_c);
        new_end->suf = eq_sbstr;
        // Make suffixes of suf_w_nxt point to eq_sbstr instead of mox_sbstr
        AutoNode *w_edge_to_eq_sbstr = suf_w_nxt;
        while (w_edge_to_eq_sbstr != 0 && w_edge_to_eq_sbstr->nxt(new_c) == max_sbstr) {
          w_edge_to_eq_sbstr->set_nxt(new_c, eq_sbstr);
          w_edge_to_eq_sbstr = w_edge_to_eq_sbstr->suf;
        }
      }
    }
    last = new_end;
  }
  SufAutomaton(string to_suffix) {
    root = new AutoNode;
    root->len = 0;
    root->suf = NULL;
    last = root;
    for (char c : to_suffix) extend(c);
  }
};
```

9 Templated multi dimensional BIT $\mathcal{O}(\log(n)^{\mathbf{dim}})$

```
// Fully overloaded any dimensional BIT, use any type for coordinates, elements, return_value.
// Includes coordinate compression.
template < typename elem_t, typename coord_t, coord_t n_inf, typename ret_t >
class BIT {
  vector< coord_t > positions;
  vector< elem_t > elems;
  bool initiated = false;

 public:
  BIT() {
    positions.push_back(n_inf);
  }
  void initiate() {
    if (initiated) {
      for (elem_t &c_elem : elems)
        c_elem.initiate();
    } else {
      initiated = true;
      sort(positions.begin(), positions.end());
      positions.resize(unique(positions.begin(), positions.end()) - positions.begin());
      elems.resize(positions.size());
    }
  }
  template < typename... loc_form >
  void update(coord_t cord, loc_form... args) {
    if (initiated) {
      int pos = lower_bound(positions.begin(), positions.end(), cord) - positions.begin();
      for (; pos < positions.size(); pos += pos & -pos)
        elems[pos].update(args...);
    } else {
      positions.push_back(cord);
    }
  }
  template < typename... loc_form >
  ret_t query(coord_t cord, loc_form... args) { //sum in open interval (-inf, cord)
```

```
    ret_t res = 0;
    int pos = (lower_bound(positions.begin(), positions.end(), cord) - positions.begin())-1;
    for (; pos > 0; pos -= pos & -pos)
      res += elems[pos].query(args...);
    return res;
  }
};
template < typename internal_type >
struct wrapped {
  internal_type a = 0;
  void update(internal_type b) {
    a += b;
  }
  internal_type query() {
    return a;
  }
  // Should never be called, needed for compilation
  void initiate() {
    cerr << 'i' << endl;
  }
  void update() {
    cerr << 'u' << endl;
  }
};
int main() {
  // retun type should be same as type inside wrapped
  BIT< BIT< wrapped< ll >, int, INT_MIN, ll >, int, INT_MIN, ll > fenwick;
  int dim = 2;
  vector< tuple< int, int, ll > > to_insert;
  to_insert.emplace_back(1, 1, 1);
  // set up all positions that are to be used for update
  for (int i = 0; i < dim; ++i) {
    for (auto &cur : to_insert)
      fenwick.update(get< 0 >(cur), get< 1 >(cur));  // May include value which won't be used
    fenwick.initiate();
  }
  // actual use
  for (auto &cur : to_insert)
    fenwick.update(get< 0 >(cur), get< 1 >(cur), get< 2 >(cur));
  cout << fenwick.query(2, 2)<<'\n';
}
```

10 Templated HLD $\mathcal{O}(M(n)\log n)$ per query

```
class dummy {
public:
  dummy () {
  }

  dummy (int, int) {
  }

  void set (int, int) {
  }

  int query (int left, int right) {
    cout << this << ' ' << left << ' ' << right << endl;
  }
};

/* T should be the type of the data stored in each vertex;
 * DS should be the underlying data structure that is used to peform the
 * group operation. It should have the following methods:
 * * DS () - empty constructor
 * * DS (int size, T initial) - constructs the structure with the given size,
 *   initially filled with initial.
 * * void set (int index, T value) - set the value at index `index` to `value`
 * * T query (int left, int right) - return the "sum" of elements between left and right, inclusive.
 */
template<typename T, class DS>
class HLD {
  int vertexc;
  vector<int> *adj;
```

```
  vector<int> subtree_size;
  DS structure;
  DS aux;

  void build_sizes (int vertex, int parent) {
    subtree_size[vertex] = 1;
    for (int child : adj[vertex]) {
      if (child != parent) {
        build_sizes(child, vertex);
        subtree_size[vertex] += subtree_size[child];
      }
    }
  }

  int cur;
  vector<int> ord;
  vector<int> chain_root;
  vector<int> par;
  void build_hld (int vertex, int parent, int chain_source) {
    cur++;
    ord[vertex] = cur;
    chain_root[vertex] = chain_source;
    par[vertex] = parent;

    if (adj[vertex].size() > 1) {
      int big_child, big_size = -1;
      for (int child : adj[vertex]) {
        if ((child != parent) &&
            (subtree_size[child] > big_size)) {
          big_child = child;
          big_size = subtree_size[child];
        }
      }

      build_hld(big_child, vertex, chain_source);
      for (int child : adj[vertex]) {
        if ((child != parent) && (child != big_child)) {
          build_hld(child, vertex, child);
        }
      }
    }
  }

public:
  HLD (int _vertexc) {
    vertexc = _vertexc;
    adj = new vector<int> [vertexc + 5];
  }

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

  void build (T initial) {
    subtree_size = vector<int> (vertexc + 5);
    ord = vector<int> (vertexc + 5);
    chain_root = vector<int> (vertexc + 5);
    par = vector<int> (vertexc + 5);
    cur = 0;
    build_sizes(1, -1);
    build_hld(1, -1, 1);
    structure = DS (vertexc + 5, initial);
    aux = DS (50, initial);
  }

  void set (int vertex, int value) {
    structure.set(ord[vertex], value);
  }

  T query_path (int u, int v) { /* returns the "sum" of the path u->v */
    int cur_id = 0;
    while (chain_root[u] != chain_root[v]) {
```

```
      if (ord[u] > ord[v]) {
        cur_id++;
        aux.set(cur_id, structure.query(ord[chain_root[u]], ord[u]));
        u = par[chain_root[u]];
      } else {
        cur_id++;
        aux.set(cur_id, structure.query(ord[chain_root[v]], ord[v]));
        v = par[chain_root[v]];
      }
    }

    cur_id++;
    aux.set(cur_id, structure.query(min(ord[u], ord[v]), max(ord[u], ord[v])));

    return aux.query(1, cur_id);
  }

  void print () {
    for (int i = 1; i <= vertexc; i++) {
      cout << i << ' ' << ord[i] << ' ' << chain_root[i] << ' ' << par[i] << endl;
    }
  }
};

int main () {
  int vertexc;
  cin >> vertexc;

  HLD<int, dummy> hld (vertexc);
  for (int i = 0; i < vertexc - 1; i++) {
    int u, v;
    cin >> u >> v;

    hld.add_edge(u, v);
  }
  hld.build(0);
  hld.print();

  int queryc;
  cin >> queryc;
  for (int i = 0; i < queryc; i++) {
    int u, v;
    cin >> u >> v;

    hld.query_path(u, v);
    cout << endl;
  }
}
```

11 Templated Persitent Segment Tree $\mathcal{O}(\log n)$ per query

```
template<typename T, typename comp>
class PersistentST {
  struct Node {
    Node *left, *right;
    int lend, rend;
    T value;

    Node (int position, T _value) {
      left = NULL;
      right = NULL;
      lend = position;
      rend = position;
      value = _value;
    }

    Node (Node *_left, Node *_right) {
      left = _left;
      right = _right;
      lend = left->lend;
      rend = right->rend;
      value = comp()(left->value, right->value);
    }
```

```

    T query (int qleft, int qright) {
      qleft = max(qleft, lend);
      qright = min(qright, rend);

      if (qleft == lend && qright == rend) {
        return value;
      } else if (qleft > qright) {
        return comp().identity;
      } else {
        return comp()(left->query(qleft, qright),
                      right->query(qleft, qright));
      }
    }
  };

  int size;
  Node **tree;
  vector<Node*> roots;
public:
  PersistentST () {
  }

  PersistentST (int _size, T initial) {
    for (int i = 0; i < 32; i++) {
      if ((1 << i) > _size) {
        size = 1 << i;
        break;
      }
    }

    tree = new Node* [2 * size + 5];

    for (int i = size; i < 2 * size; i++) {
      tree[i] = new Node (i - size, initial);
    }

    for (int i = size - 1; i > 0; i--) {
      tree[i] = new Node (tree[2 * i], tree[2 * i + 1]);
    }

    roots = vector<Node*> (1, tree[1]);
  }

  void set (int position, T _value) {
    tree[size + position] = new Node (position, _value);
    for (int i = (size + position) / 2; i >= 1; i /= 2) {
      tree[i] = new Node (tree[2 * i], tree[2 * i + 1]);
    }
    roots.push_back(tree[1]);
  }

  int last_revision () {
    return (int) roots.size() - 1;
  }

  T query (int qleft, int qright, int revision) {
    return roots[revision]->query(qleft, qright);
  }

  T query (int qleft, int qright) {
    return roots[last_revision()]->query(qleft, qright);
  }
};
```

12 FFT $\mathcal{O}(n \log(n))$

```
//Assumes a is a power of two
vector<complex<long double> > fastFourierTransform(vector<complex<long double> > a, bool inverse) {
  const long double PI = acos(-1.0L);
  int n = a.size();
  //Precalculate w
  vector<complex<long double> > w(n, 0.0L);
```

```
  w[0] = 1;
  for(int tpow = 1; tpow < n; tpow *= 2)
    w[tpow] = polar(1.0L, 2*PI * tpow/n * (inverse ? -1 : 1) );
  for(int i=3, last = 2;i<n;i++) {
    if(w[i] == 0.0L)
      w[i] = w[last] * w[i-last];
    else
      last = i;
  }

  //Rearrange a
  for(int block = n; block > 1; block /= 2) {
    int half = block/2;
    vector<complex<long double> > na(n);
    for(int s=0; s < n; s += block)
      for(int i=0;i<block;i++)
        na[s + half*(i%2) + i/2] = a[s+i];
    a = na;
  }

  //Now do the calculation
  for(int block = 2; block <= n; block *= 2) {
    vector<complex<long double> > na(n);
    int wb = n/block, half = block/2;

    for(int s=0; s < n; s += block)
      for(int i=0;i<half; i++) {
        na[s+i] = a[s+i] + w[wb*i] * a[s+half+i];
        na[s+half+i] = a[s+i] - w[wb*i] * a[s+half+i];
      }
    a = na;
  }

  return a;
}


struct Polynomial {
  vector<long double> a;

  long double& operator[](int ind) {
    return a[ind];
  }

  Polynomial& operator*=(long double r) {
    for(auto &c : a)
      c *= r;
    return *this;
  }
  Polynomial operator*(long double r) {return Polynomial(*this) *= r;}

  Polynomial& operator/=(long double r) {
    for(auto &c : a)
      c /= r;
    return *this;
  }
  Polynomial operator/(long double r) {return Polynomial(*this) /= r;}

  Polynomial& operator+=(Polynomial r) {
    if(a.size() < r.a.size())
      a.resize(r.a.size(), 0.0L);
    for(int i=0;i<(int)r.a.size();i++)
      a[i] += r[i];
    return *this;
  }
  Polynomial operator+(Polynomial r) {return Polynomial(*this) += r;}

  Polynomial& operator-=(Polynomial r) {
    if(a.size() < r.a.size())
      a.resize(r.a.size(), 0.0L);
    for(int i=0;i<(int)r.a.size();i++)
      a[i] -= r[i];
    return *this;
```

```
  }
  Polynomial operator-(Polynomial r) {return Polynomial(*this) -= r;}

  Polynomial operator*(Polynomial r) {
    int n = 1;
    while(n < (int)(a.size() + r.a.size() - 1) )
      n *= 2;

    vector<complex<long double> > fl(n, 0.0L), fr(n, 0.0L);
    for(int i=0;i<(int)a.size();i++)
      fl[i] = a[i];
    for(int i=0;i<(int)r.a.size();i++)
      fr[i] = r[i];

    fl = fastFourierTransform(fl, false);
    fr = fastFourierTransform(fr, false);

    vector<complex<long double> > ret(n);
    for(int i=0;i<n;i++)
      ret[i] = fl[i] * fr[i];
    ret = fastFourierTransform(ret, true);

    Polynomial result;
    result.a.resize(a.size() + r.a.size() - 1);
    for(int i=0;i<(int)result.a.size();i++)
      result[i] = ret[i].real() / n;
    return result;
  }
};
```

13 MOD int, extended Euctlidean

```
pair<int, int> extendedEuclideanAlgorithm(int a, int b) {
  if(b == 0)
    return make_pair(1, 0);
  pair<int, int> ret = extendedEuclideanAlgorithm(b, a%b);
  return {ret.second, ret.first - a/b * ret.second};
}


struct Modint {
  static const int MOD = 1000000007;
  int val;

  Modint(int nval = 0) {
    val = nval;
  }

  Modint& operator+=(Modint r) {
    val = (val + r.val) % MOD;
    return *this;
  }
  Modint operator+(Modint r) {return Modint(*this) += r;}

  Modint& operator-=(Modint r) {
    val = (val + MOD - r.val) % MOD;
    return *this;
  }
  Modint operator-(Modint r) {return Modint(*this) -= r;}

  Modint& operator*=(Modint r) {
    val = 1LL * val * r.val % MOD;
    return *this;
  }
  Modint operator*(Modint r) {return Modint(*this) *= r;}

  Modint inverse() {
    int ret = extendedEuclideanAlgorithm(val, MOD).first;
    if(ret < 0)
      ret += MOD;
    return ret;
  }

```

```
  Modint& operator/=(Modint r) {
    return operator*=(r.inverse() );
  }
  Modint operator/(Modint r) {return Modint(*this) /= r;}
};
```

14 Factsheet

Combinatorics Cheat Sheet

Useful formulas

$\binom{n}{k} = \frac{n!}{k!(n-k)!}$ — number of ways to choose k objects out of n

$\binom{n+k-1}{k-1}$ — number of ways to choose k objects out of n with repetitions

$\left[{n \atop m}\right]$ — Stirling numbers of the first kind; number of permutations of n elements with k cycles

$\left[{n+1 \atop m}\right] = n\left[{n \atop m}\right] + \left[{n \atop m-1}\right]$

$(x)_n = x(x-1)\ldots x-n+1 = \sum\limits_{k=0}^{n}(-1)^{n-k}\left[{n \atop k}\right]x^k$

$\left\{{n \atop m}\right\}$ — Stirling numbers of the second kind; number of partitions of set $1,\ldots,n$ into k disjoint subsets.

$\left\{{n+1 \atop m}\right\} = k\left\{{n \atop k}\right\} + \left\{{n \atop k-1}\right\}$

$\sum\limits_{k=0}^{n}\left\{{n \atop k}\right\}(x)_k = x^n$

$C_n = \frac{1}{n+1}\binom{2n}{n}$ — Catalan numbers

$C(x) = \frac{1-\sqrt{1-4x}}{2x}$

Binomial transform

If $a_n = \sum\limits_{k=0}^{n}\binom{n}{k}b_k$, then $b_n = \sum\limits_{k=0}^{n}(-1)^{n-k}\binom{n}{k}a_k$

- $a = (1, x, x^2, \ldots), b = (1, (x+1), (x+1)^2, \ldots)$
- $a_i = i^k, b_i = \left\{{n \atop i}\right\}i!$

Burnside's lemma

Let G be a group of *action* on set X (Ex.: cyclic shifts of array, rotations and symmetries of $n \times n$ matrix, ...)

Call two objects x and y *equivalent* if there is an action f that transforms x to y: $f(x) = y$.

The number of equivalence classes then can be calculated as follows: $C = \frac{1}{|G|}\sum\limits_{f \in G}|X^f|$, where X^f is the set of *fixed points* of f: $X^f = \{x | f(x) = x\}$

Generating functions

Ordinary generating function (o.g.f.) for sequence $a_0, a_1, \ldots, a_n, \ldots$ is $A(x) = \sum\limits_{n=0}^{\infty}a_i x^i$

Exponential generating function (e.g.f.) for sequence $a_0, a_1, \ldots, a_n, \ldots$ is $A(x) = \sum\limits_{n=0}^{\infty}a_i x^i$

$B(x) = A'(x), b_{n-1} = n \cdot a_n$

$c_n = \sum\limits_{k=0}^{n}a_k b_{n-k}$ (o.g.f. convolution)

$c_n = \sum\limits_{k=0}^{n}\binom{n}{k}a_k b_{n-k}$ (e.g.f. convolution, compute with FFT using $\widetilde{a_n} = \frac{a_n}{n!}$)

General linear recurrences

If $a_n = \sum\limits_{k=1}^{n}b_k a_{n-k}$, then $A(x) = \frac{a_0}{1-B(x)}$. We also can compute all a_n with Divide-and-Conquer algorithm in $O(n \log^2 n)$.

Inverse polynomial modulo x^l

Given $A(x)$, find $B(x)$ such that $A(x)B(x) = 1 + x^l \cdot Q(x)$ for some $Q(x)$

1. Start with $B_0(x) = \frac{1}{a_0}$
2. Double the length of $B(x)$: $B_{k+1}(x) = (-B_k(x)^2 A(x) + 2B_k(x)) \mod x^{2^{k+1}}$

Fast subset convolution

Given array a_i of size 2^k, calculate $b_i = \sum\limits_{j \& i = i} b_j$

```
for b = 0..k-1
  for i = 0..2^k-1
    if (i & (1 << b)) != 0:
      a[i + (1 << b)] += a[i]
```

Hadamard transform

Treat array a of size 2^k as k-dimentional array of size $2 \times 2 \times \ldots \times 2$, calculate FFT of that array:

```
for b = 0..k-1
  for i = 0..2^k-1
    if (i & (1 << b)) != 0:
      u = a[i], v = a[i + (1 << b)]
      a[i] = u + v
      a[i + (1 << b)] = u - v
```

- **Fermat's little theorem.** Let p be prime. Then, for each integer a:

$$a^{p-1} \equiv 1 \pmod{p}.$$

Thus:

$$a^k \equiv a^{k \bmod (p-1)} \pmod{p}.$$

Also:

$$a^{p-2} \equiv a^{-1} \pmod{p}.$$

- **Iterating over subsets.** Let `mask` be the binary representation of a set. Then `for (int i = mask; i != 0; i = (i - 1) & mask)` will iterate over all the nonempty subsets of `mask`.
- **Chinese remainder theorem.** We know that:

$$\begin{aligned} x &\equiv a_1 \pmod{n_1} \\ x &\equiv a_2 \pmod{n_2} \end{aligned}$$

where n_1 and n_2 are (co)prime. We want to find $a_{1,2}$ so that:

$$x \equiv a_{1,2} \pmod{n_1 \cdot n_2}.$$

A solution is given by:

$$a_{1,2} = a_1 m_2 n_2 + a_2 m_1 n_1,$$

where m_1 and m_2 are integers so that $m_1 n_1 + m_2 n_2 = 1$. Those values can be found using the Extended Euclidean algorithm.
- **Sum of harmonic series.**

$$\frac{1}{1} + \frac{1}{2} + \cdots + \frac{1}{n} \in \mathcal{O}(\log n)$$

- **Number of primes below...**

10^2	25
10^3	168
10^4	1229
10^5	9592
10^6	78498
10^7	664579