CU-Later Code Library

yangster67, ikaurov, serichaoo

May 21th 2024

Contents

Templates

Ken's template

```cpp
#include <bits/stdc++.h>
using namespace std;
#define all(v) (v).begin(), (v).end()
typedef long long ll;
typedef long double ld;
#define pb push_back
#define sz(x) (int)(x).size()
#define fi first
#define se second
#define endl '\n'
```

Kevin's template

```cpp
// paste Kaurov's Template, minus last line
typedef vector<int> vi;
typedef vector<ll> vll;
typedef pair<int, int> pii;
typedef pair<ll, ll> pll;
typedef pair<double, double> pdd;
const ld PI = acosl(-1);
const ll mod7 = 1e9 + 7;
const ll mod9 = 998244353;
const ll INF = 2*1024*1024*1023;
const char nl = '\n';
#define forn(i, n) for (int i = 0; i < int(n); i++)
ll k, n, m, u, v, w;
string s, t;

bool multiTest = 1;
void solve(int tt){
}

int main(){
  ios::sync_with_stdio(0);cin.tie(0);cout.tie(0);
  cout<<fixed<< setprecision(14);

  int t = 1;
  if (multiTest) cin >> t;
  forn(ii, t) solve(ii);
}
```

Kevin's Template Extended

- to type after the start of the contest

```cpp
#pragma GCC target("avx2,bmi,bmi2,lzcnt,popcnt")
#include <ext/pb_ds/assoc_container.hpp>
#include <ext/pb_ds/tree_policy.hpp>
using namespace __gnu_pbds;
template<class T> using ordered_set = tree<T, null_type,
 ↪ less<T>, rb_tree_tag, tree_order_statistics_node_update>;
vi d4x = {1, 0, -1, 0};
vi d4y = {0, 1, 0, -1};
vi d8x = {1, 0, -1, 0, 1, 1, -1, -1};
vi d8y = {0, 1, 0, -1, 1, -1, 1, -1};
mt19937
 ↪ rng(chrono::steady_clock::now().time_since_epoch().count());
```

Geometry

```cpp
template<typename T>
struct TPoint{
  T x, y;
  int id;
  static constexpr T eps = static_cast<T>(1e-9);
  TPoint() : x(0), y(0), id(-1) {}
  TPoint(const T& x_, const T& y_) : x(x_), y(y_), id(-1) {}
  TPoint(const T& x_, const T& y_, const int id_) : x(x_),
 ↪ y(y_), id(id_) {}

  TPoint operator + (const TPoint& rhs) const {
```

```
    return TPoint(x + rhs.x, y + rhs.y);
  }
  TPoint operator - (const TPoint& rhs) const {
    return TPoint(x - rhs.x, y - rhs.y);
  }
  TPoint operator * (const T& rhs) const {
    return TPoint(x * rhs, y * rhs);
  }
  TPoint operator / (const T& rhs) const {
    return TPoint(x / rhs, y / rhs);
  }
  TPoint ort() const {
    return TPoint(-y, x);
  }
  T abs2() const {
    return x * x + y * y;
  }
};
template<typename T>
bool operator< (TPoint<T>& A, TPoint<T>& B){
  return make_pair(A.x, A.y) < make_pair(B.x, B.y);
}
template<typename T>
bool operator== (TPoint<T>& A, TPoint<T>& B){
  return abs(A.x - B.x) <= TPoint<T>::eps && abs(A.y - B.y) <=
  ↪ TPoint<T>::eps;
}
template<typename T>
struct TLine{
  T a, b, c;
  TLine() : a(0), b(0), c(0) {}
  TLine(const T& a_, const T& b_, const T& c_) : a(a_), b(b_),
  ↪ c(c_) {}
  TLine(const TPoint<T>& p1, const TPoint<T>& p2){
    a = p1.y - p2.y;
    b = p2.x - p1.x;
    c = -a * p1.x - b * p1.y;
  }
};
template<typename T>
T det(const T& a11, const T& a12, const T& a21, const T& a22){
  return a11 * a22 - a12 * a21;
}
template<typename T>
T sq(const T& a){
  return a * a;
}
template<typename T>
T smul(const TPoint<T>& a, const TPoint<T>& b){
  return a.x * b.x + a.y * b.y;
}
template<typename T>
T vmul(const TPoint<T>& a, const TPoint<T>& b){
  return det(a.x, a.y, b.x, b.y);
}
template<typename T>
bool parallel(const TLine<T>& l1, const TLine<T>& l2){
  return abs(vmul(TPoint<T>(l1.a, l1.b), TPoint<T>(l2.a,
  ↪ l2.b))) <= TPoint<T>::eps;
}
template<typename T>
bool equivalent(const TLine<T>& l1, const TLine<T>& l2){
  return parallel(l1, l2) &&
  abs(det(l1.b, l1.c, l2.b, l2.c)) <= TPoint<T>::eps &&
  abs(det(l1.a, l1.c, l2.a, l2.c)) <= TPoint<T>::eps;
}
template<typename T>
TPoint<T> intersection(const TLine<T>& l1, const TLine<T>&
  ↪ l2){
  return TPoint<T>(
    det(-l1.c, l1.b, -l2.c, l2.b) / det(l1.a, l1.b, l2.a,
  ↪ l2.b),
    det(l1.a, -l1.c, l2.a, -l2.c) / det(l1.a, l1.b, l2.a,
  ↪ l2.b)
  );
}
template<typename T>
int sign(const T& x){
  if (abs(x) <= TPoint<T>::eps) return 0;
  return x > 0? +1 : -1;
}
template<typename T>
T area(const vector<TPoint<T>>& pts){
  int n = sz(pts);
  T ans = 0;
  for (int i = 0; i < n; i++){
    ans += vmul(pts[i], pts[(i + 1) % n]);
  }
  return abs(ans) / 2;
}
template<typename T>
T dist_pp(const TPoint<T>& a, const TPoint<T>& b){
  return sqrt(sq(a.x - b.x) + sq(a.y - b.y));
}
template<typename T>
TLine<T> perp_line(const TLine<T>& l, const TPoint<T>& p){
  T na = -l.b, nb = l.a, nc = - na * p.x - nb * p.y;
  return TLine<T>(na, nb, nc);
}
template<typename T>
TPoint<T> projection(const TPoint<T>& p, const TLine<T>& l){
  return intersection(l, perp_line(l, p));
}
template<typename T>
T dist_pl(const TPoint<T>& p, const TLine<T>& l){
  return dist_pp(p, projection(p, l));
}
template<typename T>
struct TRay{
  TLine<T> l;
  TPoint<T> start, dirvec;
  TRay() : l(), start(), dirvec() {}
  TRay(const TPoint<T>& p1, const TPoint<T>& p2){
    l = TLine<T>(p1, p2);
    start = p1, dirvec = p2 - p1;
  }
};
template<typename T>
bool is_on_line(const TPoint<T>& p, const TLine<T>& l){
  return abs(l.a * p.x + l.b * p.y + l.c) <= TPoint<T>::eps;
}
template<typename T>
bool is_on_ray(const TPoint<T>& p, const TRay<T>& r){
  if (is_on_line(p, r.l)){
    return sign(smul(r.dirvec, TPoint<T>(p - r.start))) != -1;
  }
  else return false;
}
template<typename T>
bool is_on_seg(const TPoint<T>& P, const TPoint<T>& A, const
  ↪ TPoint<T>& B){
  return is_on_ray(P, TRay<T>(A, B)) && is_on_ray(P,
  ↪ TRay<T>(B, A));
}
template<typename T>
T dist_pr(const TPoint<T>& P, const TRay<T>& R){
  auto H = projection(P, R.l);
  return is_on_ray(H, R)? dist_pp(P, H) : dist_pp(P, R.start);
}
template<typename T>
T dist_ps(const TPoint<T>& P, const TPoint<T>& A, const
  ↪ TPoint<T>& B){
  auto H = projection(P, TLine<T>(A, B));
  if (is_on_seg(H, A, B)) return dist_pp(P, H);
  else return min(dist_pp(P, A), dist_pp(P, B));
}
template<typename T>
bool acw(const TPoint<T>& A, const TPoint<T>& B){
  T mul = vmul(A, B);
  return mul > 0 || abs(mul) <= TPoint<T>::eps;
}
template<typename T>
bool cw(const TPoint<T>& A, const TPoint<T>& B){
  T mul = vmul(A, B);
```

```
  return mul < 0 || abs(mul) <= TPoint<T>::eps;
}
template<typename T>
vector<TPoint<T>> convex_hull(vector<TPoint<T>> pts){
  sort(all(pts));
  pts.erase(unique(all(pts)), pts.end());
  vector<TPoint<T>> up, down;
  for (auto p : pts){
    while (sz(up) > 1 && acw(up.end()[-1] - up.end()[-2], p - up.end()[-2])) up.pop_back();
    while (sz(down) > 1 && cw(down.end()[-1] - down.end()[-2], p - down.end()[-2])) down.pop_back();
    up.pb(p), down.pb(p);
  }
  for (int i = sz(up) - 2; i >= 1; i--) down.pb(up[i]);
  return down;
}

template<typename T>
bool in_triangle(TPoint<T>& P, TPoint<T>& A, TPoint<T>& B, TPoint<T>& C){
  if (is_on_seg(P, A, B) || is_on_seg(P, B, C) || is_on_seg(P, C, A)) return true;
  return cw(P - A, B - A) == cw(P - B, C - B) &&
  cw(P - A, B - A) == cw(P - C, A - C);
}
template<typename T>
void prep_convex_poly(vector<TPoint<T>>& pts){
  rotate(pts.begin(), min_element(all(pts)), pts.end());
}
// 0 - Outside, 1 - Exclusively Inside, 2 - On the Border
template<typename T>
int in_convex_poly(TPoint<T>& p, vector<TPoint<T>>& pts){
  int n = sz(pts);
  if (!n) return 0;
  if (n <= 2) return is_on_seg(p, pts[0], pts.back());
  int l = 1, r = n - 1;
  while (r - l > 1){
    int mid = (l + r) / 2;
    if (acw(pts[mid] - pts[0], p - pts[0])) l = mid;
    else r = mid;
  }
  if (!in_triangle(p, pts[0], pts[l], pts[l + 1])) return 0;
  if (is_on_seg(p, pts[l], pts[l + 1]) ||
    is_on_seg(p, pts[0], pts.back()) ||
    is_on_seg(p, pts[0], pts[1])
  ) return 2;
  return 1;
}
// 0 - Outside, 1 - Exclusively Inside, 2 - On the Border
template<typename T>
int in_simple_poly(TPoint<T> p, vector<TPoint<T>>& pts){
  int n = sz(pts);
  bool res = 0;
  for (int i = 0; i < n; i++){
    auto a = pts[i], b = pts[(i + 1) % n];
    if (is_on_seg(p, a, b)) return 2;
    if (((a.y > p.y) - (b.y > p.y)) * vmul(b - p, a - p) > TPoint<T>::eps){
      res ^= 1;
    }
  }
  return res;
}
template<typename T>
void minkowski_rotate(vector<TPoint<T>>& P){
  int pos = 0;
  for (int i = 1; i < sz(P); i++){
    if (abs(P[i].y - P[pos].y) <= TPoint<T>::eps){
      if (P[i].x < P[pos].x) pos = i;
    }
    else if (P[i].y < P[pos].y) pos = i;
  }
  rotate(P.begin(), P.begin() + pos, P.end());
}
// P and Q are strictly convex, points given in counterclockwise order
template<typename T>
vector<TPoint<T>> minkowski_sum(vector<TPoint<T>> P, vector<TPoint<T>> Q){
  minkowski_rotate(P);
  minkowski_rotate(Q);
  P.pb(P[0]);
  Q.pb(Q[0]);
  vector<TPoint<T>> ans;
  int i = 0, j = 0;
  while (i < sz(P) - 1 || j < sz(Q) - 1){
    ans.pb(P[i] + Q[j]);
    T curmul;
    if (i == sz(P) - 1) curmul = -1;
    else if (j == sz(Q) - 1) curmul = +1;
    else curmul = vmul(P[i + 1] - P[i], Q[j + 1] - Q[j]);
    if (abs(curmul) < TPoint<T>::eps || curmul > 0) i++;
    if (abs(curmul) < TPoint<T>::eps || curmul < 0) j++;
  }
  return ans;
}
using Point = TPoint<ll>; using Line = TLine<ll>; using Ray = TRay<ll>; const ld PI = acos(-1);
```

Strings

```
vector<int> prefix_function(string s){
  int n = sz(s);
  vector<int> pi(n);
  for (int i = 1; i < n; i++){
    int k = pi[i - 1];
    while (k > 0 && s[i] != s[k]){
      k = pi[k - 1];
    }
    pi[i] = k + (s[i] == s[k]);
  }
  return pi;
}
vector<int> kmp(string s, string k){
  string st = k + "#" + s;
  vector<int> res;
  auto pi = pf(st);
  for (int i = 0; i < sz(st); i++){
    if (pi[i] == sz(k)){
      res.pb(i - 2 * sz(k));
    }
  }
  return res;
}
vector<int> z_function(string s){
  int n = sz(s);
  vector<int> z(n);
  int l = 0, r = 0;
  for (int i = 1; i < n; i++){
    if (r >= i) z[i] = min(z[i - l], r - 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;
    }
  }
  return z;
}
```

Graph Theory

Max Flow

```
struct Edge {
  int from, to, cap, remain;
};

struct Dinic {
  int n;
```

```
  vector<Edge> e;
  vector<vector<int>> g;
  vector<int> d, cur;
  Dinic(int _n) : n(_n), g(n), d(n), cur(n) {}
  void add_edge(int u, int v, int c) {
    g[u].push_back((int)e.size());
    e.push_back({u, v, c, c});
    g[v].push_back((int)e.size());
    e.push_back({v, u, 0, 0});
  }
  ll max_flow(int s, int t) {
    int inf = 1e9;
    auto bfs = [&]() {
      fill(d.begin(), d.end(), inf), fill(cur.begin(), cur.end(), 0);
      d[s] = 0;
      vector<int> q{s}, nq;
      for (int step = 1; q.size(); swap(q, nq), nq.clear(), step++) {
        for (auto& node : q) {
          for (auto& edge : g[node]) {
            int ne = e[edge].to;
            if (!e[edge].remain || d[ne] <= step) continue;
            d[ne] = step, nq.push_back(ne);
            if (ne == t) return true;
          }
        }
      }
      return false;
    };
    function<int(int, int)> find = [&](int node, int limit) {
      if (node == t || !limit) return limit;
      int flow = 0;
      for (int i = cur[node]; i < g[node].size(); i++) {
        cur[node] = i;
        int edge = g[node][i], oe = edge ^ 1, ne = e[edge].to;
        if (!e[edge].remain || d[ne] != d[node] + 1) continue;
        if (int temp = find(ne, min(limit - flow, e[edge].remain))) {
          e[edge].remain -= temp, e[oe].remain += temp, flow += temp;
        } else {
          d[ne] = -1;
        }
        if (flow == limit) break;
      }
      return flow;
    };
    ll res = 0;
    while (bfs())
      while (int flow = find(s, inf)) res += flow;
    return res;
  }
};
```

- USAGE

```
int main() {
  int n, m, s, t;
  cin >> n >> m >> s >> t;
  Dinic dinic(n);
  for (int i = 0, u, v, c; i < m; i++) {
    cin >> u >> v >> c;
    dinic.add_edge(u - 1, v - 1, c);
  }
  cout << dinic.max_flow(s - 1, t - 1) << '\n';
}
```

PushRelabel Max-Flow (faster)

```
//
// https://github.com/kth-competitive-programming/kactl/blob/main/content/graph/PushRelabel.h
#define rep(i, a, b) for (int i = a; i < (b); ++i)
#define all(x) begin(x), end(x)
#define sz(x) (int)(x).size()
typedef long long ll;
typedef pair<int, int> pii;
typedef vector<int> vi;

struct PushRelabel {
  struct Edge {
    int dest, back;
    ll f, c;
  };
  vector<vector<Edge>> g;
  vector<ll> ec;
  vector<Edge*> cur;
  vector<vi> hs;
  vi H;
  PushRelabel(int n) : g(n), ec(n), cur(n), hs(2 * n), H(n) {}

  void addEdge(int s, int t, ll cap, ll rcap = 0) {
    if (s == t) return;
    g[s].push_back({t, sz(g[t]), 0, cap});
    g[t].push_back({s, sz(g[s]) - 1, 0, rcap});
  }

  void addFlow(Edge& e, ll f) {
    Edge& back = g[e.dest][e.back];
    if (!ec[e.dest] && f) hs[H[e.dest]].push_back(e.dest);
    e.f += f;
    e.c -= f;
    ec[e.dest] += f;
    back.f -= f;
    back.c += f;
    ec[back.dest] -= f;
  }
  ll calc(int s, int t) {
    int v = sz(g);
    H[s] = v;
    ec[t] = 1;
    vi co(2 * v);
    co[0] = v - 1;
    rep(i, 0, v) cur[i] = g[i].data();
    for (Edge& e : g[s]) addFlow(e, e.c);

    for (int hi = 0;;) {
      while (hs[hi].empty())
        if (!hi--) return -ec[s];
      int u = hs[hi].back();
      hs[hi].pop_back();
      while (ec[u] > 0)  // discharge u
        if (cur[u] == g[u].data() + sz(g[u])) {
          H[u] = 1e9;
          for (Edge& e : g[u])
            if (e.c && H[u] > H[e.dest] + 1) H[u] = H[e.dest] + 1, cur[u] = &e;
          if (++co[H[u]], !--co[hi] && hi < v)
            rep(i, 0, v) if (hi < H[i] && H[i] < v)--
  co[H[i]], H[i] = v + 1;
          hi = H[u];
        } else if (cur[u]->c && H[u] == H[cur[u]->dest] + 1)
          addFlow(*cur[u], min(ec[u], cur[u]->c));
        else
          ++cur[u];
    }
  }
  bool leftOfMinCut(int a) { return H[a] >= sz(g); }
};
```

Min-Cost Max-Flow

```
class MCMF {
public:
  static constexpr int INF = 1e9;
  const int n;
  vector<tuple<int, int, int>> e;
  vector<vector<int>> g;
  vector<int> h, dis, pre;
  bool dijkstra(int s, int t) {
    dis.assign(n, INF);
    pre.assign(n, -1);
    priority_queue<pair<int, int>, vector<pair<int, int>>, greater<>> que;
```

```
    dis[s] = 0;
    que.emplace(0, s);
    while (!que.empty()) {
      auto [d, u] = que.top();
      que.pop();
      if (dis[u] != d) continue;
      for (int i : g[u]) {
        auto [v, f, c] = e[i];
        if (c > 0 && dis[v] > d + h[u] - h[v] + f) {
          dis[v] = d + h[u] - h[v] + f;
          pre[v] = i;
          que.emplace(dis[v], v);
        }
      }
    }
    return dis[t] != INF;
  }
  MCMF(int n) : n(n), g(n) {}
  void add_edge(int u, int v, int fee, int c) {
    g[u].push_back(e.size());
    e.emplace_back(v, fee, c);
    g[v].push_back(e.size());
    e.emplace_back(u, -fee, 0);
  }
  pair<ll, ll> max_flow(const int s, const int t) {
    int flow = 0, cost = 0;
    h.assign(n, 0);
    while (dijkstra(s, t)) {
      for (int i = 0; i < n; ++i) h[i] += dis[i];
      for (int i = t; i != s; i = get<0>(e[pre[i] ^ 1])) {
        --get<2>(e[pre[i]]);
        ++get<2>(e[pre[i] ^ 1]);
      }
      ++flow;
      cost += h[t];
    }
    return {flow, cost};
  }
};
```

Max Cost Feasible Flow

```
struct Edge {
  int from, to, cap, remain, cost;
};

struct MCMF {
  int n;
  vector<Edge> e;
  vector<vector<int>> g;
  vector<ll> d, pre;
  MCMF(int _n) : n(_n), g(n), d(n), pre(n) {}
  void add_edge(int u, int v, int c, int w) {
    g[u].push_back((int)e.size());
    e.push_back({u, v, c, c, w});
    g[v].push_back((int)e.size());
    e.push_back({v, u, 0, 0, -w});
  }
  pair<ll, ll> max_flow(int s, int t) {
    ll inf = 1e18;
    auto spfa = [&]() {
      fill(d.begin(), d.end(), -inf); // important!
      vector<int> f(n), seen(n);
      d[s] = 0, f[s] = 1e9;
      vector<int> q{s}, nq;
      for (; q.size(); swap(q, nq), nq.clear()) {
        for (auto& node : q) {
          seen[node] = false;
          for (auto& edge : g[node]) {
            int ne = e[edge].to, cost = e[edge].cost;
            if (!e[edge].remain || d[ne] >= d[node] + cost)
↪ continue;
            d[ne] = d[node] + cost, pre[ne] = edge;
            f[ne] = min(e[edge].remain, f[node]);
            if (!seen[ne]) seen[ne] = true, nq.push_back(ne);
          }
        }
      }
      return f[t];
    };
    ll flow = 0, cost = 0;
    while (int temp = spfa()) {
      if (d[t] < 0) break; // important!
      flow += temp, cost += temp * d[t];
      for (ll i = t; i != s; i = e[pre[i]].from) {
        e[pre[i]].remain -= temp, e[pre[i] ^ 1].remain +=
↪ temp;
      }
    }
    return {flow, cost};
  }
};
```

Heavy-Light Decomposition

```
struct HeavyLight {
  int root = 0, n = 0;
  std::vector<int> parent, deep, hson, top, sz, dfn;
  HeavyLight(std::vector<std::vector<int>> &g, int _root)
      : root(_root), n(int(g.size())), parent(n), deep(n),
↪ hson(n, -1), top(n), sz(n), dfn(n, -1) {
    int cur = 0;
    std::function<int(int, int, int)> dfs = [&](int node, int
↪ fa, int dep) {
      deep[node] = dep, sz[node] = 1, parent[node] = fa;
      for (auto &ne : g[node]) {
        if (ne == fa) continue;
        sz[node] += dfs(ne, node, dep + 1);
        if (hson[node] == -1 || sz[ne] > sz[hson[node]])
↪ hson[node] = ne;
      }
      return sz[node];
    };
    std::function<void(int, int)> dfs2 = [&](int node, int t)
↪ {
      top[node] = t, dfn[node] = cur++;
      if (hson[node] == -1) return;
      dfs2(hson[node], t);
      for (auto &ne : g[node]) {
        if (ne == parent[node] || ne == hson[node]) continue;
        dfs2(ne, ne);
      }
    };
    dfs(root, -1, 0), dfs2(root, root);
  }

  int lca(int x, int y) const {
    while (top[x] != top[y]) {
      if (deep[top[x]] < deep[top[y]]) swap(x, y);
      x = parent[top[x]];
    }
    return deep[x] < deep[y] ? x : y;
  }

  std::vector<std::array<int, 2>> get_dfn_path(int x, int y)
↪ const {
    std::array<std::vector<std::array<int, 2>>, 2> path;
    bool front = true;
    while (top[x] != top[y]) {
      if (deep[top[x]] > deep[top[y]]) swap(x, y), front =
↪ !front;
      path[front].push_back({dfn[top[y]], dfn[y] + 1});
      y = parent[top[y]];
    }
    if (deep[x] > deep[y]) swap(x, y), front = !front;

    path[front].push_back({dfn[x], dfn[y] + 1});
    std::reverse(path[1].begin(), path[1].end());
    for (const auto &[left, right] : path[1])
↪ path[0].push_back({right, left});
    return path[0];
  }

  Node query_seg(int u, int v, const SegTree &seg) const {
```

```
    auto node = Node();
    for (const auto &[left, right] : get_dfn_path(u, v)) {
      if (left > right) {
        node = pull(node, rev(seg.query(right, left)));
      } else {
        node = pull(node, seg.query(left, right));
      }
    }
    return node;
  }
};
```

- USAGE:

```
vector<ll> light(n);
SegTree heavy(n), form_parent(n);
// cin >> x >> y, x--, y--;
int z = lca(x, y);
while (x != z) {
  if (dfn[top[x]] <= dfn[top[z]]) {
    // [dfn[z], dfn[x]), from heavy
    heavy.modify(dfn[z], dfn[x], 1);
    break;
  }
  // x -> top[x];
  heavy.modify(dfn[top[x]], dfn[x], 1);
  light[parent[top[x]]] += a[top[x]];
  x = parent[top[x]];
}
while (y != z) {
  if (dfn[top[y]] <= dfn[top[z]]) {
    // (dfn[z], dfn[y]], from heavy
    form_parent.modify(dfn[z] + 1, dfn[y] + 1, 1);
    break;
  }
  // y -> top[y];
  form_parent.modify(dfn[top[y]], dfn[y] + 1, 1);
  y = parent[top[y]];
}
```

General Unweight Graph Matching

- Complexity: $O(n^3)$ (?)

```
struct BlossomMatch {
  int n;
  vector<vector<int>> e;
  BlossomMatch(int _n) : n(_n), e(_n) {}
  void add_edge(int u, int v) { e[u].push_back(v),
 ↪  e[v].push_back(u); }
  vector<int> find_matching() {
    vector<int> match(n, -1), vis(n), link(n), f(n), dep(n);
    function<int(int)> find = [&](int x) { return f[x] == x ?
 ↪  x : (f[x] = find(f[x])); };
    auto lca = [&](int u, int v) {
      u = find(u), v = find(v);
      while (u != v) {
        if (dep[u] < dep[v]) swap(u, v);
        u = find(link[match[u]]);
      }
      return u;
    };
    queue<int> que;
    auto blossom = [&](int u, int v, int p) {
      while (find(u) != p) {
        link[u] = v, v = match[u];
        if (vis[v] == 0) vis[v] = 1, que.push(v);
        f[u] = f[v] = p, u = link[v];
      }
    };
    // find an augmenting path starting from u and augment (if
 ↪  exist)
    auto augment = [&](int node) {
      while (!que.empty()) que.pop();
      iota(f.begin(), f.end(), 0);
      // vis = 0 corresponds to inner vertices, vis = 1
 ↪  corresponds to outer vertices
      fill(vis.begin(), vis.end(), -1);
      que.push(node);
      vis[node] = 1, dep[node] = 0;
      while (!que.empty()) {
        int u = que.front();
        que.pop();
        for (auto v : e[u]) {
          if (vis[v] == -1) {
            vis[v] = 0, link[v] = u, dep[v] = dep[u] + 1;
            // found an augmenting path
            if (match[v] == -1) {
              for (int x = v, y = u, temp; y != -1; x = temp,
 ↪  y = x == -1 ? -1 : link[x]) {
                temp = match[y], match[x] = y, match[y] = x;
              }
              return;
            }
            vis[match[v]] = 1, dep[match[v]] = dep[u] + 2;
            que.push(match[v]);
          } else if (vis[v] == 1 && find(v) != find(u)) {
            // found a blossom
            int p = lca(u, v);
            blossom(u, v, p), blossom(v, u, p);
          }
        }
      }
    };
    // find a maximal matching greedily (decrease constant)
    auto greedy = [&]() {
      for (int u = 0; u < n; ++u) {
        if (match[u] != -1) continue;
        for (auto v : e[u]) {
          if (match[v] == -1) {
            match[u] = v, match[v] = u;
            break;
          }
        }
      }
    };
    greedy();
    for (int u = 0; u < n; ++u)
      if (match[u] == -1) augment(u);
    return match;
  }
};
```

Maximum Bipartite Matching

- Needs dinic, complexity $\approx O(n + m\sqrt{n})$

```
struct BipartiteMatch {
  int l, r;
  Dinic dinic = Dinic(0);
  BipartiteMatch(int _l, int _r) : l(_l), r(_r) {
    dinic = Dinic(l + r + 2);
    for (int i = 1; i <= l; i++) dinic.add_edge(0, i, 1);
    for (int i = 1; i <= r; i++) dinic.add_edge(l + i, l + r +
 ↪  1, 1);
  }
  void add_edge(int u, int v) { dinic.add_edge(u + 1, l + v +
 ↪  1, 1); }
  ll max_matching() { return dinic.max_flow(0, l + r + 1); }
};
```

2-SAT and Strongly Connected Components

```
void scc(vector<vector<int>>& g, int* idx) {
  int n = g.size(), ct = 0;
  int out[n];
  vector<int> ginv[n];
  memset(out, -1, sizeof out);
  memset(idx, -1, n * sizeof(int));
  function<void(int)> dfs = [&](int cur) {
    out[cur] = INT_MAX;
    for(int v : g[cur]) {
      ginv[v].push_back(cur);
```

```
      if(out[v] == -1) dfs(v);
    }
    ct++; out[cur] = ct;
  };
  vector<int> order;
  for(int i = 0; i < n; i++) {
    order.push_back(i);
    if(out[i] == -1) dfs(i);
  }
  sort(order.begin(), order.end(), [&](int& u, int& v) {
    return out[u] > out[v];
  });
  ct = 0;
  stack<int> s;
  auto dfs2 = [&](int start) {
    s.push(start);
    while(!s.empty()) {
      int cur = s.top();
      s.pop();
      idx[cur] = ct;
      for(int v : ginv[cur])
        if(idx[v] == -1) s.push(v);
    }
  };
  for(int v : order) {
    if(idx[v] == -1) {
      dfs2(v);
      ct++;
    }
  }
}

// 0 => impossible, 1 => possible
pair<int,vector<int>> sat2(int n, vector<pair<int,int>>&
 ↪  clauses) {
  vector<int> ans(n);
  vector<vector<int>> g(2*n + 1);
  for(auto [x, y] : clauses) {
    x = x < 0 ? -x + n : x;
    y = y < 0 ? -y + n : y;
    int nx = x <= n ? x + n : x - n;
    int ny = y <= n ? y + n : y - n;
    g[nx].push_back(y);
    g[ny].push_back(x);
  }
  int idx[2*n + 1];
  scc(g, idx);
  for(int i = 1; i <= n; i++) {
    if(idx[i] == idx[i + n]) return {0, {}};
    ans[i - 1] = idx[i + n] < idx[i];
  }
  return {1, ans};
}
```

Enumerating Triangles

- Complexity: $O(n + m\sqrt{m})$

```
void enumerate_triangles(vector<pair<int,int>>& edges,
 ↪  function<void(int,int,int)> f) {
  int n = 0;
  for(auto [u, v] : edges) n = max({n, u + 1, v + 1});
  vector<int> deg(n);
  vector<int> g[n];
  for(auto [u, v] : edges) {
    deg[u]++;
    deg[v]++;
  }
  for(auto [u, v] : edges) {
    if(u == v) continue;
    if(deg[u] > deg[v] || (deg[u] == deg[v] && u > v))
      swap(u, v);
    g[u].push_back(v);
  }
  vector<int> flag(n);
  for(int i = 0; i < n; i++) {
    for(int v : g[i]) flag[v] = 1;
    for(int v : g[i]) for(int u : g[v]) {
      if(flag[u]) f(i, v, u);
    }
    for(int v : g[i]) flag[v] = 0;
  }
}
```

Tarjan

- shrink all circles into points (2-edge-connected-component)

```
int cnt = 0, now = 0;
vector<ll> dfn(n, -1), low(n), belong(n, -1), stk;
function<void(ll, ll)> tarjan = [&](ll node, ll fa) {
  dfn[node] = low[node] = now++, stk.push_back(node);
  for (auto& ne : g[node]) {
    if (ne == fa) continue;
    if (dfn[ne] == -1) {
      tarjan(ne, node);
      low[node] = min(low[node], low[ne]);
    } else if (belong[ne] == -1) {
      low[node] = min(low[node], dfn[ne]);
    }
  }
  if (dfn[node] == low[node]) {
    while (true) {
      auto v = stk.back();
      belong[v] = cnt;
      stk.pop_back();
      if (v == node) break;
    }
    ++cnt;
  }
};
```

- 2-vertex-connected-component / Block forest

```
int cnt = 0, now = 0;
vector<vector<ll>> e1(n);
vector<ll> dfn(n, -1), low(n), stk;
function<void(ll)> tarjan = [&](ll node) {
  dfn[node] = low[node] = now++, stk.push_back(node);
  for (auto& ne : g[node]) {
    if (dfn[ne] == -1) {
      tarjan(ne);
      low[node] = min(low[node], low[ne]);
      if (low[ne] == dfn[node]) {
        e1.push_back({});
        while (true) {
          auto x = stk.back();
          stk.pop_back();
          e1[n + cnt].push_back(x);
          // e1[x].push_back(n + cnt); // undirected
          if (x == ne) break;
        }
        e1[node].push_back(n + cnt);
        // e1[n + cnt].push_back(node); // undirected
        cnt++;
      }
    } else {
      low[node] = min(low[node], dfn[ne]);
    }
  }
};
```

Kruskal reconstruct tree

```
int _n, m;
cin >> _n >> m; // _n: # of node, m: # of edge
int n = 2 * _n - 1; // root: n-1
vector<array<int, 3>> edges(m);
for (auto& [w, u, v] : edges) {
  cin >> u >> v >> w, u--, v--;
}
sort(edges.begin(), edges.end());
```

```
vector<int> p(n);
iota(p.begin(), p.end(), 0);
function<int(int)> find = [&](int x) { return p[x] == x ? x : (p[x] = find(p[x])); };
auto merge = [&](int x, int y) { p[find(x)] = find(y); };
vector<vector<int>> g(n);
vector<int> val(m);
val.reserve(n);
for (auto [w, u, v] : edges) {
  u = find(u), v = find(v);
  if (u == v) continue;
  val.push_back(w);
  int node = (int)val.size() - 1;
  g[node].push_back(u), g[node].push_back(v);
  merge(u, node), merge(v, node);
}
```

centroid decomposition

```
vector<char> res(n), seen(n), sz(n);
function<int(int, int)> get_size = [&](int node, int fa) {
  sz[node] = 1;
  for (auto& ne : g[node]) {
    if (ne == fa || seen[ne]) continue;
    sz[node] += get_size(ne, node);
  }
  return sz[node];
};
function<int(int, int, int)> find_centroid = [&](int node, int fa, int t) {
  for (auto& ne : g[node])
    if (ne != fa && !seen[ne] && sz[ne] > t / 2) return find_centroid(ne, node, t);
  return node;
};
function<void(int, char)> solve = [&](int node, char cur) {
  get_size(node, -1); auto c = find_centroid(node, -1, sz[node]);
  seen[c] = 1, res[c] = cur;
  for (auto& ne : g[c]) {
    if (seen[ne]) continue;
    solve(ne, char(cur + 1)); // we can pass c here to build tree
  }
};
```

virtual tree

```
map<int, vector<int>> gg; vector<int> stk{0};
auto add = [&](int x, int y) { gg[x].push_back(y), gg[y].push_back(x); };
for (int i = 0; i < k; i++) {
  if (a[i] != 0) {
    int p = lca(a[i], stk.back());
    if (p != stk.back()) {
      while (dfn[p] < dfn[stk[int(stk.size()) - 2]]) {
        add(stk.back(), stk[int(stk.size()) - 2]);
        stk.pop_back();
      }
      add(p, stk.back()), stk.pop_back();
      if (dfn[p] > dfn[stk.back()]) stk.push_back(p);
    }
    stk.push_back(a[i]);
  }
}
while (stk.size() > 1) {
  if (stk.back() != 0) {
    add(stk.back(), stk[int(stk.size()) - 2]);
    stk.pop_back();
  }
}
```

Flows

O(N^2 * M), on unit networks O(N^(1/2) * M)

```
struct FlowEdge {
    int v, u;
    long long cap, flow = 0;
    FlowEdge(int v, int u, long long cap) : v(v), u(u), cap(cap) {}
};
struct Dinic {
    const long long flow_inf = 1e18;
    vector<FlowEdge> edges;
    vector<vector<int>> adj;
    int n, m = 0;
    int s, t;
    vector<int> level, ptr;
    queue<int> q;
    Dinic(int n, int s, int t) : n(n), s(s), t(t) {
        adj.resize(n);
        level.resize(n);
        ptr.resize(n);
    }
    void add_edge(int v, int u, long long cap) {
        edges.emplace_back(v, u, cap);
        edges.emplace_back(u, v, 0);
        adj[v].push_back(m);
        adj[u].push_back(m + 1);
        m += 2;
    }
    bool bfs() {
        while (!q.empty()) {
            int v = q.front();
            q.pop();
            for (int id : adj[v]) {
                if (edges[id].cap - edges[id].flow < 1)
                    continue;
                if (level[edges[id].u] != -1)
                    continue;
                level[edges[id].u] = level[v] + 1;
                q.push(edges[id].u);
            }
        }
        return level[t] != -1;
    }
    long long dfs(int v, long long pushed) {
        if (pushed == 0)
            return 0;
        if (v == t)
            return pushed;
        for (int& cid = ptr[v]; cid < (int)adj[v].size(); cid++) {
            int id = adj[v][cid];
            int u = edges[id].u;
            if (level[v] + 1 != level[u] || edges[id].cap - edges[id].flow < 1)
                continue;
            long long tr = dfs(u, min(pushed, edges[id].cap - edges[id].flow));
            if (tr == 0)
                continue;
            edges[id].flow += tr;
            edges[id ^ 1].flow -= tr;
            return tr;
        }
        return 0;
    }
    long long flow() {
        long long f = 0;
        while (true) {
            fill(level.begin(), level.end(), -1);
            level[s] = 0;
            q.push(s);
            if (!bfs())
                break;
```

```
            fill(ptr.begin(), ptr.end(), 0);
            while (long long pushed = dfs(s, flow_inf)) {
                f += pushed;
            }
        }
        return f;
    }
};
// To recover flow through original edges: iterate over even indices in edges.
```

MCMF – maximize flow, then minimize its cost. O(F*mn*).

```
#include <ext/pb_ds/priority_queue.hpp>
template <typename T, typename C>
class MCMF {
 public:
  static constexpr T eps = (T) 1e-9;

  struct edge {
    int from;
    int to;
    T c;
    T f;
    C cost;
  };

  int n;
  vector<vector<int>> g;
  vector<edge> edges;
  vector<C> d;
  vector<C> pot;
  __gnu_pbds::priority_queue<pair<C, int>> q;
  vector<typename decltype(q)::point_iterator> its;
  vector<int> pe;
  const C INF_C = numeric_limits<C>::max() / 2;

  explicit MCMF(int n_) : n(n_), g(n), d(n), pot(n, 0), its(n), pe(n) {}

  int add(int from, int to, T forward_cap, C edge_cost, T backward_cap = 0) {
    assert(0 <= from && from < n && 0 <= to && to < n);
    assert(forward_cap >= 0 && backward_cap >= 0);
    int id = static_cast<int>(edges.size());
    g[from].push_back(id);
    edges.push_back({from, to, forward_cap, 0, edge_cost});
    g[to].push_back(id + 1);
    edges.push_back({to, from, backward_cap, 0, -edge_cost});
    return id;
  }

  void expath(int st) {
    fill(d.begin(), d.end(), INF_C);
    q.clear();
    fill(its.begin(), its.end(), q.end());
    its[st] = q.push({pot[st], st});
    d[st] = 0;
    while (!q.empty()) {
      int i = q.top().second;
      q.pop();
      its[i] = q.end();
      for (int id : g[i]) {
        const edge &e = edges[id];
        int j = e.to;
        if (e.c - e.f > eps && d[i] + e.cost < d[j]) {
          d[j] = d[i] + e.cost;
          pe[j] = id;
          if (its[j] == q.end()) {
            its[j] = q.push({pot[j] - d[j], j});
          } else {
            q.modify(its[j], {pot[j] - d[j], j});
          }
        }
      }
    }
    swap(d, pot);
  }

  pair<T, C> max_flow(int st, int fin) {
    T flow = 0;
    C cost = 0;
    bool ok = true;
    for (auto& e : edges) {
      if (e.c - e.f > eps && e.cost + pot[e.from] - pot[e.to] < 0) {
        ok = false;
        break;
      }
    }
    if (ok) {
      expath(st);
    } else {
      vector<int> deg(n, 0);
      for (int i = 0; i < n; i++) {
        for (int eid : g[i]) {
          auto& e = edges[eid];
          if (e.c - e.f > eps) {
            deg[e.to] += 1;
          }
        }
      }
      vector<int> que;
      for (int i = 0; i < n; i++) {
        if (deg[i] == 0) {
          que.push_back(i);
        }
      }
      for (int b = 0; b < (int) que.size(); b++) {
        for (int eid : g[que[b]]) {
          auto& e = edges[eid];
          if (e.c - e.f > eps) {
            deg[e.to] -= 1;
            if (deg[e.to] == 0) {
              que.push_back(e.to);
            }
          }
        }
      }
      fill(pot.begin(), pot.end(), INF_C);
      pot[st] = 0;
      if (static_cast<int>(que.size()) == n) {
        for (int v : que) {
          if (pot[v] < INF_C) {
            for (int eid : g[v]) {
              auto& e = edges[eid];
              if (e.c - e.f > eps) {
                if (pot[v] + e.cost < pot[e.to]) {
                  pot[e.to] = pot[v] + e.cost;
                  pe[e.to] = eid;
                }
              }
            }
          }
        }
      } else {
        que.assign(1, st);
        vector<bool> in_queue(n, false);
        in_queue[st] = true;
        for (int b = 0; b < (int) que.size(); b++) {
          int i = que[b];
          in_queue[i] = false;
          for (int id : g[i]) {
            const edge &e = edges[id];
            if (e.c - e.f > eps && pot[i] + e.cost < pot[e.to]) {
              pot[e.to] = pot[i] + e.cost;
              pe[e.to] = id;
              if (!in_queue[e.to]) {
                que.push_back(e.to);
                in_queue[e.to] = true;
              }
```

```
            }
          }
        }
      }
    }
    while (pot[fin] < INF_C) {
      T push = numeric_limits<T>::max();
      int v = fin;
      while (v != st) {
        const edge &e = edges[pe[v]];
        push = min(push, e.c - e.f);
        v = e.from;
      }
      v = fin;
      while (v != st) {
        edge &e = edges[pe[v]];
        e.f += push;
        edge &back = edges[pe[v] ^ 1];
        back.f -= push;
        v = e.from;
      }
      flow += push;
      cost += push * pot[fin];
      expath(st);
    }
    return {flow, cost};
  }
};

// Examples: MCMF<int, int> g(n); g.add(u,v,c,w,0);
 ↪  g.max_flow(s,t).
// To recover flow through original edges: iterate over even
 ↪  indices in edges.
```

Graphs

Kuhn's algorithm for bipartite matching

```
/*
The graph is split into 2 halves of n1 and n2 vertices.
Complexity: O(n1 * m). Usually runs much faster. MUCH
 ↪  FASTER!!!
*/
const int N = 305;

vector<int> g[N]; // Stores edges from left half to right.
bool used[N]; // Stores if vertex from left half is used.
int mt[N]; // For every vertex in right half, stores to which
 ↪  vertex in left half it's matched (-1 if not matched).

bool try_dfs(int v){
  if (used[v]) return false;
  used[v] = 1;
  for (auto u : g[v]){
    if (mt[u] == -1 || try_dfs(mt[u])){
      mt[u] = v;
      return true;
    }
  }
  return false;
}

int main(){
// ......
  for (int i = 1; i <= n2; i++) mt[i] = -1;
  for (int i = 1; i <= n1; i++) used[i] = 0;
  for (int i = 1; i <= n1; i++){
    if (try_dfs(i)){
      for (int j = 1; j <= n1; j++) used[j] = 0;
    }
  }
  vector<pair<int, int>> ans;
  for (int i = 1; i <= n2; i++){
    if (mt[i] != -1) ans.pb({mt[i], i});
  }
}
```

```

// Finding maximal independent set: size = # of nodes - # of
 ↪  edges in matching.
// To construct: launch Kuhn-like DFS from unmatched nodes in
 ↪  the left half.
// Independent set = visited nodes in left half + unvisited in
 ↪  right half.
// Finding minimal vertex cover: complement of maximal
 ↪  independent set.
```

Dijkstra's Algorithm

```
priority_queue<pair<ll, ll>, vector<pair<ll, ll>>,
 ↪  greater<pair<ll, ll>>> q;
dist[start] = 0;
q.push({0, start});
while (!q.empty()){
    auto [d, v] = q.top();
    q.pop();
    if (d != dist[v]) continue;
    for (auto [u, w] : g[v]){
      if (dist[u] > dist[v] + w){
        dist[u] = dist[v] + w;
        q.push({dist[u], u});
      }
    }
}
```

EULERIAN CYCLE DFS

```
void dfs(int v){
  while (!g[v].empty()){
    int u = g[v].back();
    g[v].pop_back();
    dfs(u);
    ans.pb(v);
  }
}
```

Strongly Connected Components: Kosaraju's Algorithm

```
vector<vector<int>> adj, adj_rev;
vector<bool> used;
vector<int> order, component;

void dfs1(int v) {
    used[v] = true;

    for (auto u : adj[v])
        if (!used[u])
            dfs1(u);

    order.push_back(v);
}

void dfs2(int v) {
    used[v] = true;
    component.push_back(v);

    for (auto u : adj_rev[v])
        if (!used[u])
            dfs2(u);
}

int main(){
// ......
    used.assign(n, false);

    for (int i = 0; i < n; i++)
        if (!used[i])
            dfs1(i);
    used.assign(n, false);
    reverse(order.begin(), order.end());
    for (auto v : order)
```

```
        if (!used[v]) {
            dfs2(v);
            // process
            component.clear();
        }
}
```

Finding Bridges

```
/*
Bridges.
Results are stored in a map "is_bridge".
For each connected component, call "dfs(starting vertex,
 ↪  starting vertex)".
*/
const int N = 2e5 + 10; // Careful with the constant!

vector<int> g[N];
int tin[N], fup[N], timer;
map<pair<int, int>, bool> is_bridge;

void dfs(int v, int p){
  tin[v] = ++timer;
  fup[v] = tin[v];
  for (auto u : g[v]){
    if (!tin[u]){
      dfs(u, v);
      if (fup[u] > tin[v]){
        is_bridge[{u, v}] = is_bridge[{v, u}] = true;
      }
      fup[v] = min(fup[v], fup[u]);
    }
    else{
      if (u != p) fup[v] = min(fup[v], tin[u]);
    }
  }
}
```

Virtual Tree

```
// order stores the nodes in the queried set
sort(all(order), [&] (int u, int v){return tin[u] < tin[v];});
int m = sz(order);
for (int i = 1; i < m; i++){
    order.pb(lca(order[i], order[i - 1]));
}
sort(all(order), [&] (int u, int v){return tin[u] < tin[v];});
order.erase(unique(all(order)), order.end());
vector<int> stk{order[0]};
for (int i = 1; i < sz(order); i++){
    int v = order[i];
    while (tout[stk.back()] < tout[v]) stk.pop_back();
    int u = stk.back();
    vg[u].pb({v, dep[v] - dep[u]});
    stk.pb(v);
}
```

HLD ON EDGES DFS

```
void dfs1(int v, int p, int d){
  par[v] = p;
  for (auto e : g[v]){
    if (e.fi == p){
      g[v].erase(find(all(g[v]), e));
      break;
    }
  }
  dep[v] = d;
  sz[v] = 1;
  for (auto [u, c] : g[v]){
    dfs1(u, v, d + 1);
    sz[v] += sz[u];
  }
  if (!g[v].empty()) iter_swap(g[v].begin(),
 ↪  max_element(all(g[v]), comp));
}
void dfs2(int v, int rt, int c){
  pos[v] = sz(a);
  a.pb(c);
  root[v] = rt;
  for (int i = 0; i < sz(g[v]); i++){
    auto [u, c] = g[v][i];
    if (!i) dfs2(u, rt, c);
    else dfs2(u, u, c);
  }
}
int getans(int u, int v){
  int res = 0;
  for (; root[u] != root[v]; v = par[root[v]]){
    if (dep[root[u]] > dep[root[v]]) swap(u, v);
    res = max(res, rmq(0, 0, n - 1, pos[root[v]], pos[v]));
  }
  if (pos[u] > pos[v]) swap(u, v);
  return max(res, rmq(0, 0, n - 1, pos[u] + 1, pos[v]));
}
```

Centroid Decomposition

```
vector<char> res(n), seen(n), sz(n);
function<int(int, int)> get_size = [&](int node, int fa) {
  sz[node] = 1;
  for (auto& ne : g[node]) {
    if (ne == fa || seen[ne]) continue;
    sz[node] += get_size(ne, node);
  }
  return sz[node];
};
function<int(int, int, int)> find_centroid = [&](int node, int
 ↪  fa, int t) {
  for (auto& ne : g[node])
    if (ne != fa && !seen[ne] && sz[ne] > t / 2) return
 ↪  find_centroid(ne, node, t);
  return node;
};
function<void(int, char)> solve = [&](int node, char cur) {
  get_size(node, -1); auto c = find_centroid(node, -1,
 ↪  sz[node]);
  seen[c] = 1, res[c] = cur;
  for (auto& ne : g[c]) {
    if (seen[ne]) continue;
    solve(ne, char(cur + 1)); // we can pass c here to build
 ↪  tree
  }
};
```