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;
}
```

Flows

$O(N^2M)$, 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) {}
};
```

```cpp
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. $O(Fmn)$.

```cpp
#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
  }
};
```

Number Theory

EXTENDED EUCLIDEAN ALGORITHM

```
// gives (x, y) for ax + by = g
// solutions given (x0, y0): a(x0 + kb/g) + b(y0 - ka/g) = g
int gcd(int a, int b, int& x, int& y) {
  x = 1, y = 0; int sum1 = a;
  int x2 = 0, y2 = 1, sum2 = b;
  while (sum2) {
    int q = sum1 / sum2;
    tie(x, x2) = make_tuple(x2, x - q * x2);
    tie(y, y2) = make_tuple(y2, y - q * y2);
    tie(sum1, sum2) = make_tuple(sum2, sum1 - q * sum2);
  }
  return sum1;
}
```

Linear Sieve

```
vector<int> prime;
bool is_composite[MAX_N];
int phi[MAX_N];

void sieve (int n){
  fill(is_composite, is_composite + n, false);
  phi[1] = 1;
  for (int i = 2; i < n; ++i) {
    if (!is_composite[i]) {
      prime.push_back (i);
      phi[i] = i - 1; //i is prime
      }
  for (int j = 0; j < prime.size () && i * prime[j] < n; ++j)
↪ {
    is_composite[i * prime[j]] = true;
    if (i % prime[j] == 0) {
      phi[i * prime[j]] = phi[i] * prime[j]; //prime[j]
↪ divides i
      break;
      } else {
      phi[i * prime[j]] = phi[i] * phi[prime[j]]; //prime[j]
↪ does not divide i
      }
    }
  }
}
```

Data Structures

Sparse Table

```
const int N = 2e5 + 10, LOG = 20; // Change the constant!
template<typename T>
struct SparseTable{
int lg[N];
T st[N][LOG];
int n;

// Change this function
function<T(T, T)> f = [&] (T a, T b){
  return min(a, b);
};

void build(vector<T>& a){
  n = sz(a);
  lg[1] = 0;
  for (int i = 2; i <= n; i++) lg[i] = lg[i / 2] + 1;

  for (int k = 0; k < LOG; k++){
    for (int i = 0; i < n; i++){
      if (!k) st[i][k] = a[i];
      else st[i][k] = f(st[i][k - 1], st[min(n - 1, i + (1 <<
↪ (k - 1)))][k - 1]);
    }
  }
}

T query(int l, int r){
  int sz = r - l + 1;
  return f(st[l][lg[sz]], st[r - (1 << lg[sz]) + 1][lg[sz]]);
}
};
```

Suffix Array and LCP array

- (uses SparseTable above)

```
struct SuffixArray{
  vector<int> p, c, h;
  SparseTable<int> st;
  /*
  In the end, array c gives the position of each suffix in p
  using 1-based indexation!
  */

  SuffixArray() {}

  SuffixArray(string s){
    buildArray(s);
    buildLCP(s);
    buildSparse();
  }

  void buildArray(string s){
    int n = sz(s) + 1;
    p.resize(n), c.resize(n);
    for (int i = 0; i < n; i++) p[i] = i;
```

```
    sort(all(p), [&] (int a, int b){return s[a] < s[b];});
    c[p[0]] = 0;
    for (int i = 1; i < n; i++){
      c[p[i]] = c[p[i - 1]] + (s[p[i]] != s[p[i - 1]]);
    }
    vector<int> p2(n), c2(n);
    // w is half-length of each string.
    for (int w = 1; w < n; w <<= 1){
      for (int i = 0; i < n; i++){
        p2[i] = (p[i] - w + n) % n;
      }
      vector<int> cnt(n);
      for (auto i : c) cnt[i]++;
      for (int i = 1; i < n; i++) cnt[i] += cnt[i - 1];
      for (int i = n - 1; i >= 0; i--){
        p[--cnt[c[p2[i]]]] = p2[i];
      }
      c2[p[0]] = 0;
      for (int i = 1; i < n; i++){
        c2[p[i]] = c2[p[i - 1]] +
        (c[p[i]] != c[p[i - 1]] ||
        c[(p[i] + w) % n] != c[(p[i - 1] + w) % n]);
      }
      c.swap(c2);
    }
    p.erase(p.begin());
  }

  void buildLCP(string s){
    // The algorithm assumes that suffix array is already
    ↪ built on the same string.
    int n = sz(s);
    h.resize(n - 1);
    int k = 0;
    for (int i = 0; i < n; i++){
      if (c[i] == n){
        k = 0;
        continue;
      }
      int j = p[c[i]];
      while (i + k < n && j + k < n && s[i + k] == s[j + k])
    ↪ k++;
      h[c[i] - 1] = k;
      if (k) k--;
    }
    /*
    Then an RMQ Sparse Table can be built on array h
    to calculate LCP of 2 non-consecutive suffixes.
    */
  }

  void buildSparse(){
    st.build(h);
  }

  // l and r must be in 0-BASED INDEXATION
  int lcp(int l, int r){
    l = c[l] - 1, r = c[r] - 1;
    if (l > r) swap(l, r);
    return st.query(l, r - 1);
  }
};
```

Aho Corasick Trie

- For each node in the trie, the suffix link points to the longest proper suffix of the represented string. The terminal-link tree has square-root height (can be constructed by DFS).

```
const int S = 26;

// Function converting char to int.
int ctoi(char c){
  return c - 'a';
}

// To add terminal links, use DFS
struct Node{
  vector<int> nxt;
  int link;
  bool terminal;

  Node() {
    nxt.assign(S, -1), link = 0, terminal = 0;
  }
};

vector<Node> trie(1);

// add_string returns the terminal vertex.
int add_string(string& s){
  int v = 0;
  for (auto c : s){
    int cur = ctoi(c);
    if (trie[v].nxt[cur] == -1){
      trie[v].nxt[cur] = sz(trie);
      trie.emplace_back();
    }
    v = trie[v].nxt[cur];
  }
  trie[v].terminal = 1;
  return v;
}

/*
Suffix links are compressed.
This means that:
  If vertex v has a child by letter x, then:
    trie[v].nxt[x] points to that child.
  If vertex v doesn't have such child, then:
    trie[v].nxt[x] points to the suffix link of that child
    if we would actually have it.
*/
void add_links(){
  queue<int> q;
  q.push(0);
  while (!q.empty()){
    auto v = q.front();
    int u = trie[v].link;
    q.pop();
    for (int i = 0; i < S; i++){
      int& ch = trie[v].nxt[i];
      if (ch == -1){
        ch = v? trie[u].nxt[i] : 0;
      }
      else{
        trie[ch].link = v? trie[u].nxt[i] : 0;
        q.push(ch);
      }
    }
  }
}

bool is_terminal(int v){
  return trie[v].terminal;
}

int get_link(int v){
  return trie[v].link;
}

int go(int v, char c){
  return trie[v].nxt[ctoi(c)];
}
```

Convex Hull Trick

- Allows to insert a linear function to the hull in (1) and get the minimum/maximum value of the stored function at a point in O(log n).
- NOTE: The lines must be added in the order of decreas-

ing/increasing gradients. CAREFULLY CHECK THE SETUP BEFORE USING!
- IMPORTANT: THE DEFAULT VERSION SURELY WORKS. IF MODIFIED VERSIONS DON'T WORK, TRY TRANSFORMING THEM TO THE DEFAULT ONE BY CHANGING SIGNS.

```
struct line{
  ll k, b;
  ll f(ll x){
    return k * x + b;
  };
};

vector<line> hull;

void add_line(line nl){
  if (!hull.empty() && hull.back().k == nl.k){
    nl.b = min(nl.b, hull.back().b); // Default: minimum. For maximum change "min" to "max".
    hull.pop_back();
  }
  while (sz(hull) > 1){
    auto& l1 = hull.end()[-2], l2 = hull.back();
    if ((nl.b - l1.b) * (l2.k - nl.k) >= (nl.b - l2.b) * (l1.k - nl.k)) hull.pop_back(); // Default: decreasing gradient k. For increasing k change the sign to <=.
    else break;
  }
  hull.pb(nl);
}

ll get(ll x){
  int l = 0, r = sz(hull);
  while (r - l > 1){
    int mid = (l + r) / 2;
    if (hull[mid - 1].f(x) >= hull[mid].f(x)) l = mid; // Default: minimum. For maximum change the sign to <=.
    else r = mid;
  }
  return hull[l].f(x);
}
```

Li-Chao Segment Tree

- allows to add linear functions in any order and query minimum/maximum value of those at a point, all in O(log n).
- Clear: clear()

```
const ll INF = 1e18; // Change the constant!
struct LiChaoTree{
  struct line{
    ll k, b;
    line(){
      k = b = 0;
    };
    line(ll k_, ll b_){
      k = k_, b = b_;
    };
    ll f(ll x){
      return k * x + b;
    };
  };
  int n;
  bool minimum, on_points;
  vector<ll> pts;
  vector<line> t;

  void clear(){
    for (auto& l : t) l.k = 0, l.b = minimum? INF : -INF;
  }

  LiChaoTree(int n_, bool min_){ // This is a default constructor for numbers in range [0, n - 1].
    n = n_, minimum = min_, on_points = false;
    t.resize(4 * n);
    clear();
  };

  LiChaoTree(vector<ll> pts_, bool min_){ // This constructor will build LCT on the set of points you pass. The points may be in any order and contain duplicates.
    pts = pts_, minimum = min_;
    sort(all(pts));
    pts.erase(unique(all(pts)), pts.end());
    on_points = true;
    n = sz(pts);
    t.resize(4 * n);
    clear();
  };

  void add_line(int v, int l, int r, line nl){
    // Adding on segment [l, r)
    int m = (l + r) / 2;
    ll lval = on_points? pts[l] : l, mval = on_points? pts[m] : m;
    if ((minimum && nl.f(mval) < t[v].f(mval)) || (!minimum && nl.f(mval) > t[v].f(mval))) swap(t[v], nl);
    if (r - l == 1) return;
    if ((minimum && nl.f(lval) < t[v].f(lval)) || (!minimum && nl.f(lval) > t[v].f(lval))) add_line(2 * v + 1, l, m, nl);
    else add_line(2 * v + 2, m, r, nl);
  }

  ll get(int v, int l, int r, int x){
    int m = (l + r) / 2;
    if (r - l == 1) return t[v].f(on_points? pts[x] : x);
    else{
      if (minimum) return min(t[v].f(on_points? pts[x] : x), x < m? get(2 * v + 1, l, m, x) : get(2 * v + 2, m, r, x));
      else return max(t[v].f(on_points? pts[x] : x), x < m? get(2 * v + 1, l, m, x) : get(2 * v + 2, m, r, x));
    }
  }

  void add_line(ll k, ll b){
    add_line(0, 0, n, line(k, b));
  }

  ll get(ll x){
    return get(0, 0, n, on_points? lower_bound(all(pts), x) - pts.begin() : x);
  }; // Always pass the actual value of x, even if LCT is on points.
};
```

Persistent Segment Tree

- for RSQ

```
struct Node {
    ll val;
    Node *l, *r;

    Node(ll x) : val(x), l(nullptr), r(nullptr) {}
    Node(Node *ll, Node *rr) {
        l = ll, r = rr;
        val = 0;
        if (l) val += l->val;
        if (r) val += r->val;
    }
    Node(Node *cp) : val(cp->val), l(cp->l), r(cp->r) {}
};
const int N = 2e5 + 20;
ll a[N];
Node *roots[N];
int n, cnt = 1;
Node *build(int l = 1, int r = n) {
    if (l == r) return new Node(a[l]);
    int mid = (l + r) / 2;
    return new Node(build(l, mid), build(mid + 1, r));
```

```
}
Node *update(Node *node, int val, int pos, int l = 1, int r = n) {
    if (l == r) return new Node(val);
    int mid = (l + r) / 2;
    if (pos > mid)
        return new Node(node->l, update(node->r, val, pos, mid + 1, r));
    else return new Node(update(node->l, val, pos, l, mid), node->r);
}
ll query(Node *node, int a, int b, int l = 1, int r = n) {
    if (l > b || r < a) return 0;
    if (l >= a && r <= b) return node->val;
    int mid = (l + r) / 2;
    return query(node->l, a, b, l, mid) + query(node->r, a, b, mid + 1, r);
}
```

Miscellaneous

Ordered Set

```
#include <ext/pb_ds/assoc_container.hpp>
#include <ext/pb_ds/tree_policy.hpp>
using namespace __gnu_pbds;
typedef tree<int, null_type, less<int>, rb_tree_tag, tree_order_statistics_node_update> ordered_set;
```

Measuring Execution Time

```
ld tic = clock();
// execute algo…
ld tac = clock();
// Time in milliseconds
cerr << (tac - tic) / CLOCKS_PER_SEC * 1000 << endl;
// No need to comment out the print because it's done to cerr.
```

Setting Fixed D.P. Precision

```
cout << setprecision(d) << fixed;
// Each number is rounded to d digits after the decimal point, and truncated.
```