Team Note of National Potato

lim551(unkoob), qse1346(vidm), ventulus95

Compiled on November 7, 2019

Contents

ALL BELOW HERE ARE USELESS IF YOU READ THE STATEMENT WRONG

1 Network Flow

1.1 Dinic's Algorithm

```
/*
  1. build level graph with bfs ( O(E) )
  2. flow blocking flow in level graph ( O(VE) )
  In each step, level grows at least by 1, and eventually grows upto O(V)
  So total time complexity is O(V^2E)
*/
struct dinic { // O(V^2 E)
  struct edg { int v, c, r; };
  int n;
  vi dis, itr;
  vector<vector<edg>> g;
  dinic(int n) : n(n), g(n, vector<edg>()), dis(n), itr(n) {  }
  void addedge(int u, int v, int c) {
    g[u].pb({ v, c, sz(g[v]) });
    g[v].pb({ u, 0, sz(g[u])-1 });
  }
  bool bfs(int s, int t) { // build level graph
    dis.assign(n, 0), itr.assign(n, 0);
    queue<int> q;
    q.push(s);
    dis[s] = 1;
    while(q.size()) {
      int u = q.front(); q.pop();
      for(auto& [v, c, r] : g[u]) {
        if(c > 0 && !dis[v]) {
          dis[v] = dis[u] + 1;
          q.push(v);
        }
      }
    }
    return dis[t] > 0;
  }
  int dfs(int u, int t, int f) { // get blocking flow
    if(u == t) return f;
    for( ; itr[u] < g[u].size(); itr[u]++) {
      auto& [v, c, r] = g[u][itr[u]];
      if(c > 0 && dis[v] == dis[u] + 1) {
        int w = dfs(v, t, min(f, c));
        if(w) {
          g[u][itr[u]].c -= w;
          g[v][r].c += w;
          return w;
        }
      }
    }
    return 0;
  }
  i64 nflow(int s, int t) { // network flow
    i64 ret = 0;
    while(bfs(s, t)) {
      int r;
      while(r = dfs(s, t, 2e9)) ret += r, debug("-----");
    }
    return ret;
  }
};
```

1.2 Hofcroft-Karp Bipartite Matching

```
/*
    - alternating path: path consists of (x, y, x, y, x) for X = { x | x is matched edge}, Y
= { y | y is not mathed edge }
    - augmenting path: path consists of (y, x, y, x, y)
    - if augmenting path exists, we can match one more edge with fliping matched state (x,
y, x, y, x)

    For maximaum matching A, B
    1. lv[0] = { v | v in A and v is not matched }
    2. starting from lv[0] vertices, get alternating path with bfs
    3. starting from lv[0] vertices, get augmenting path with dfs

    * min cover: selecting minimum vertices to cover all edges
    * max independent set: selecting maximum vertices not connected with edge
    * V - min cover = max independent set
*/
struct hofcroft {
  int n, m;
  vi dis, l, r, vis, chk;
  vvi g;
  hofcroft(int n, int m) : n(n), m(m), g(n, vi()) {  }
  void addedge(int u, int v) { g[u].pb(v); }
  bool bfs() { // build alternating path starts from lv[0] nodes
    queue<int> q;
    bool ok = 0;
    dis.assign(n, 0);
    FOR(u, 0, n) {
      if(l[u] == -1 && !dis[u]) {
        q.push(u);
        dis[u] = 1;
      }
    }
    while(q.size()) {
      int u = q.front(); q.pop();
      for(int v : g[u]) {
        if(r[v] == -1) ok = 1; // v is not matched
        else if(!dis[r[v]]) { // if v is matched, u>v>r[v] can be path
          dis[r[v]] = dis[u] + 1;
          q.push(r[v]);
        }
      }
    }
    return ok;
  }
  bool dfs(int u) { // find augmenting path and flip it!
    if(vis[u]) return 0; // augmenting path start/end with non-matched vertices
    vis[u] = 1;
    for(int v : g[u]) {
      if(r[v] == -1 || (dis[r[v]] == dis[u] + 1 && dfs(r[v]))) {
        l[u] = v; r[v] = u;
        return 1;
```

```
      }
    }
    return 0;
  }
  int match() { // bipartite match
    l.assign(n, -1);
    r.assign(m, -1);
    int ret = 0;
    while(bfs()) {
      vis.assign(n, 0);
      FOR(u, 0, n) if(l[u] == -1 && dfs(u)) ++ret;
    }
    return ret;
  }
  void rdfs(int u) { // dfs matched
    if(chk[u]) return;
    chk[u] = 1;
    for(int v : g[u]) {
      chk[v + n] = 1;
      rdfs(r[v]);
    }
  }
  vi getcover() { // get min cover vertices
    match();
    chk.assign(n+m, 0);
    FOR(u, 0, n) if(l[u] == -1) rdfs(u);
    vi ret;
    FOR(u, 0, n) if(!chk[u]) ret.pb(u);
    FOR(u, n, n+m) if(chk[u]) ret.pb(u);
    return ret;
  }
};
```

1.3 Minimum Cost Maximum Flow

```
const int WINF = 0x3fffffff, FINF = 0x3fffffff; // weight/flow inf
struct mcmf {
  struct edg { int v, c, r, w; };
  int n;
  vi dis, par, peg;
  vector<bool> inq;
  vector<vector<edg> > g;
  mcmf(int n) : n(n), g(n, vector<edg>()), par(n), peg(n) {  }
  void addedge(int u, int v, int c, int w) {
    g[u].pb({ v, c, sz(g[v]), w });
    g[v].pb({ u, 0, sz(g[u])-1, -w });
  }
  bool spfa(int s, int t) {
    dis.assign(n, WINF);
    inq.assign(n, 0);
    queue<int> q;
    dis[s] = 0;
    inq[s] = 1;
    q.push(s);
    bool ok = 0;
    while(q.size()) {
      int u = q.front(); q.pop();
      if(u == t) ok = 1;
      inq[u] = 0;
      FOR(eidx, 0, g[u].size()) {
        auto [v, c, r, w] = g[u][eidx];
        if(c > 0 && dis[v] > dis[u] + w) {
          dis[v] = dis[u] + w;
          par[v] = u;
          peg[v] = eidx;
          if(!inq[v]) {
            inq[v] = 1;
            q.push(v);
          }
        }
      }
    }
    return ok;
  }
  ii flow(int s, int t) { // return (max_flow, min_cost)
    int cost = 0, flow = 0;
    while(spfa(s, t)) {
      int cur = FINF;
      for(int u = t; u != s; u = par[u]) cur = min(cur, g[par[u]][peg[u]].c);
      for(int u = t; u != s; u = par[u]) {
        int r = g[par[u]][peg[u]].r;
        g[par[u]][peg[u]].c -= cur;
        g[u][r].c += cur;
      }
      flow += cur;
      cost += dis[t] * cur;
    }
    return { flow, cost };
  }
};
```

1.4 Ford-Fulkerson Algorithm

```
// Caution: All vertices' idx > 0 (par[S] = 0)
const int MAXND = 500, S = 1, T = 2, INF = 0x3fffffff;
struct nflow {
  struct edge {
    edge* rev;
    int v, c, f; // need initialized
    edge(int v, int c) : v(v), c(c), f(0) { }
    int res() { return c-f; }
    int flow(int x) { f += x, rev->f -= x; }
  };
  vector<edge*> g[MAXND];
  int par[MAXND];
  edge* pedg[MAXND];
  nflow() {
    FOR(i, 0, MAXN) g[i] = vector<edge*>();
  }
  void addedge(int u, int v, int c) {
    edge *uv = new edge(v, c), *vu = new edge(u, 0);
    uv->rev = vu, vu->rev = uv;
    g[u].pb(uv), g[v].pb(vu);
  }
  i64 maxflow() {
    i64 ret = 0;
```

```
    while(true) {
      memset(par, 0, sizeof(par));
      queue<int> q;
      q.push(S);
      par[S] = S;
      while(q.size()) {
        int u = q.front(); q.pop();
        for(auto e : g[u]) {
          if(e->res() && !par[e->v]) {
            q.push(e->v);
            par[e->v] = u;
            pedg[e->v] = e;
            if(e->v == T) break;
          }
        }
        if(par[T]) break;
      }
      if(!par[T]) break;
      int flow = INF;
      for(int u = T; u != S; u = par[u]) flow = min(flow, pedg[u]->res());
      for(int u = T; u != S; u = par[u]) pedg[u]->flow(flow);
      ret += flow;
    }
    return ret;
  }
};
```

1.5 Naive Bipartite Match

```
const int MAXN = 5e2+10;
int vis[MAXN], ato[MAXN], bto[MAXN];
bool dfs(int u) {
  if(vis[u]) return 0;
  vis[u] = 1;
  for(int v : g[u]) {
    if(bto[v] == -1 || dfs(bto[v])) {
      ato[u] = v;
      bto[v] = u;
      return 1;
    }
  }
  return 0;
}
int bimatch() {
  memset(ato, -1, sizeof(ato)), memset(bto, -1, sizeof(bto));
  int ret = 0;
  FOR(u, 0, n) {
    memset(vis, 0, sizeof(vis));
    if(dfs(u)) ++ret;
  }
  return ret;
}
```

2 Graph

2.1 2-SAT + SCC

```
/*
  2-SAT: (A || B) && (C || D) && (E || F) ...
  1. X || Y = !X -> Y, !Y -> X  (Proposition)
    False: T -> F, True: Others
  2. !X, X in same SCC: no solution
  3. For every SCC, each node in same SCC must have same flag (if both T, F exists in same
SCC, T->F exists)
  4. Assign False to (Don't have in edge & Unassigned node) and erase node
    - sort nodes topologically, iterate nodes with assigning False to var if var is
unassigned
      - !X node: X = True, X node: X = False
*/
struct sat2 {
  struct tarjan {
    int n, ncnt, scnt;
    vi scc, dis;
    vvi g;
    stack<int> sta;
    tarjan(int n) : n(n), g(n, vi()) { } // n: number of variables (NOT NODES!)
    void addedge(int u, int v) { g[u].pb(v); } // directed graph
    int f(int u) {
      int ret = dis[u] = ncnt++;
      sta.push(u);
      for(int v : g[u]) {
        if(dis[v] == -1) ret = min(ret, f(v));
        else if(scc[v] == -1) ret = min(ret, dis[v]);
      }
      if(ret == dis[u]) {
        while(1) {
          int t = sta.top(); sta.pop();
          scc[t] = scnt;
          if(t == u) break;
        }
        ++scnt;
      }
      return ret;
    }
    vi& get_scc() {
      ncnt = scnt = 0;
      scc = dis = vi(n, -1);
      sta = stack<int>();
      FOR(i, 0, n) if(dis[i] == -1) f(i);
      dis.clear();
      return scc;
    }
  };
  int n;
  vi res;
  tarjan tj;
  sat2(int n) : n(n), tj(2*n) { }
  int nd(int u, int neg) { return u + neg*n; } // var u's node
  int neg(int u) { return (u+n)%(2*n); } // ~u
  void addedge(int u, int nu, int v, int nv) { // add (X || Y) clauses
    u = nd(u, nu), v = nd(v, nv);
    tj.addedge(neg(u), v);
    tj.addedge(neg(v), u);
  }
  vi& solve() { // return solved vars, if no solution return vi()
    vi& scc = tj.get_scc();
```

```
    FOR(u, 0, n) if(scc[u] == scc[u+n]) return res;
    res.assign(n, -1);
    vi ord(2*n);
    FOR(i, 0, 2*n) ord[i] = i;
    sort(ALL(ord), [&](int u, int v) { return scc[u] > scc[v]; });
    FOR(i, 0, 2*n) {
      int u = ord[i];
      if(res[u%n] == -1) res[u%n] = !(u<n);
    }
    return res;
  }
};
```

2.2 Cut-Vertex

```
// Find Bridge
const int MAXN = 3e5, INF = 0x7fffffff;
int ncnt, vid[MAXN], par[MAXN];
vii g[MAXN]; // (v, bridge flag)
int dfs(int u) {
  int ret = vid[u] = ncnt++;
  for(auto& e : g[u]) {
    int v = e.se, c = INF;
    if(par[u] == v) continue; // Tree edge
    if(vid[v] == -1) { // Tree Edge
      par[v] = u;
      c = min(c, dfs(v));
    } else c = min({ c, vid[v], vid[u] }); // Forward/Backward Edge
    if(c > vid[u]) e.fi = 1; // Bridge
    ret = min(ret, c);
  }
  return ret;
}
```

3 String

3.1 KMP Algorithm

```
string s, t;
vi getpi(const string& str) {
  int n = str.size(), len = 0;
  vi pi(n, 0);
  FOR(i, 1, n) {
    while(len && str[len] != str[i]) len = pi[len-1];
    if(str[len] == str[i]) pi[i] = ++len;
  }
  return pi;
}

vi kmp() {
  int n = s.size(), m = t.size(), len = 0;
  vi ret, pi = getpi(t);
  FOR(i, 0, n) {
    while(len && s[i] != t[len]) len = pi[len-1];
    if(s[i] == t[len] && ++len == m) ret.pb(i-len+1), len = pi[len-1];
  }
  return ret;
}
```

3.2 Rabin-Karp Algorithm

```
// f(p) = s[0] + s[1] * p + s[2] * p^2 + ... + s[n-1] * p^{n-1}
// h[i+1] = p * (h[i] - s[i] * s^(m-1)) + s[i+m]
// --> sub first character from hash > hash degree up > add last character to hash
const i64 MUL = 232153, MD = 1012924417; // be careful for MD not MOD
void mod(i64& x) { x %= MD; if(x < 0) x += MD; }
vi rabin(string& s, string& t) { // return start indexs
  vi ret;
  i64 ht = 0, hs = 0, mul = 1;
  RFOR(i, sz(t)-1, 0) { // get t's hash
    mod(ht += mul * t[i] % MD);
    mod(mul = mul * MUL);
  }
  mul = 1;
  RFOR(i, sz(t)-1, 0) { // get s's hash for first sz(t) string
    mod(hs += mul * s[i] % MD);
    if(i != 0) mod(mul = mul * MUL); // mul must be p^{m-1}
  }
  if(hs == ht) ret.pb(0);
  FOR(i, sz(t), sz(s)) {
    mod(hs -= mul*s[i-sz(t)]%MD);
    mod(hs *= MUL);
    mod(hs += s[i]);
    if(hs == ht) ret.pb(i-sz(t)+1);
  }
  return ret;
}
```

3.3 Trie(Array)

```
const int MAX_NODE = 1e6+10;
int cld[MAX_NODE][30];
i64 cnt[MAX_NODE];
int ncnt = 1;
void push(const string& x) {
  int u = 0;
  FOR(i, 0, x.size()) {
    if(!cld[u][x[i]-'a']) cld[u][x[i]-'a'] = ncnt++;
    u = cld[u][x[i]-'a'];
  }
  ++cnt[u];
}
void calc_back(int u) {
  FOR(i, 0, 30) {
    if(cld[u][i]) {
      calc_back(cld[u][i]);
      cnt[u] += cnt[cld[u][i]];
    }
  }
  cnt[u] %= MOD;
}
```

3.4 Aho-Corasick Algorithm

```
const int MAXNODE = 1e5+10, MAXC = 26, INITCHAR = 'a';
struct ahocorasick {
  int ncnt, t[MAXNODE][MAXC], f[MAXNODE], chk[MAXNODE];
  ahocorasick() : ncnt(0) { memset(t, 0, sizeof(t)), memset(f, 0, sizeof(f)), memset(chk, 0,
  sizeof(chk)); }
```

```
  void insert(const string& s) {
    int u = 0;
    for(auto i : s) {
      i -= INITCHAR;
      if(!t[u][i]) t[u][i] = ++ncnt;
      u = t[u][i];
    }
    chk[u] = 1; // end of string
  }
  void precalc() {
    queue<int> q;
    FOR(i, 0, MAXC) if(t[0][i]) q.push(t[0][i]);
    // calculdate fail, chk with bfs
    while(q.size()) {
      int x = q.front(); q.pop();
      FOR(i, 0, MAXC) {
        if(t[x][i]) {
          int u = x, p = f[u];
          while(p && !t[p][i]) p = f[p]; // find fail link
          u = t[u][i], p = t[p][i]; // goto original target node
          f[u] = p;
          if(chk[p]) chk[u] = 1;
          q.push(u);
        }
      }
    }
  }
  bool query(const string& s) {
    int u = 0;
    for(auto i : s) {
      i -= INITCHAR;
      while(u && !t[u][i]) u = f[u];
      if(chk[u = t[u][i]]) return true;
    }
    return false;
  }
};
```

3.5 Suffix Array + LCP

```
/*
  sa[i] = ordered suffix (suffix's start position)
  ord[i] = [i:]'s index in sa  (ord[sa[i]] = i)
  lcp[i] = longest common prefix length of two suffix [i-1:], [i:]

  LCP's Lemma
  1.  Two adjacent in SA suffixes' LCP is always bigger than which of non-adjacents
  2.  lcp(sa[i-1], sa[i]) = h, h >= 1 then
    lcp(sa[i-1]+1, sa[i]+1) = h-1

  So that lcp[sa[i]+1] >= h-1 because it is always bigger than lcp(sa[i-1]+1, sa[i]+1) by
Lemma 1
  and by Lemma 2, lcp(sa[i-1]+1, sa[i]+1) = h-1
*/
struct sfxarray {
  int n;
  string& str;
  vi sa, lcp, ord;
  sfxarray(string& str) : str(str), n(str.size()) { }
  void getsa() {
    sa = ord = vi(n+1);
    FOR(i, 0, n) sa[i] = i, ord[i] = str[i]; ord[n] = 0;
    for(int t = 1; t <= n; t *= 2) {
      int sz = max(257, n+1);
      vi cnt, tmp;
      cnt = tmp = vi(sz, 0);
      FOR(i, 0, n) ++cnt[ord[min(n, i+t)]];
      FOR(i, 1, sz) cnt[i] += cnt[i-1];
      FOR(i, 0, n) tmp[--cnt[ord[min(n, i+t)]]] = i;
      cnt = vi(sz, 0);
      FOR(i, 0, n) ++cnt[ord[i]];
      FOR(i, 1, sz) cnt[i] += cnt[i-1];
      RFOR(i, n-1, 0) sa[--cnt[ord[tmp[i]]]] = tmp[i];
      tmp[sa[0]] = 1;
      FOR(i, 1, n) {
        int u = sa[i-1], v = sa[i];
        tmp[v] = tmp[u] + (ord[u] < ord[v] || ord[u+t] < ord[v+t]);
      }
      ord = tmp;
      if(ord[sa[n-1]] == n) break;
    }
    FOR(i, 0, n) --ord[i];
  }
  void getlcp() {
    lcp = vi(n, 0);
    for(int i = 0, len = 0; i < n; ++i, len = max(0, len-1)) {
      if(ord[i]) {
        for(int j = sa[ord[i]-1]; str[i+len] == str[j+len]; ++len);
        lcp[ord[i]] = len;
      }
    }
  }
  tuple<vi, vi, vi> build() { getsa(), getlcp(); return { sa, lcp, ord }; }
};
```

3.6 Manacher's Algorithm

```
const int MAXN = 1000005;
int aux[2 * MAXN - 1];
void solve(int n, int *str, int *ret){
  // *ret : number of nonobvious palindromic character pair
  for(int i=0; i<n; i++){
    aux[2*i] = str[i];
    if(i != n-1) aux[2*i+1] = -1;
  }
  int p = 0, c = 0;
  for(int i=0; i<2*n-1; i++){
    int cur = 0;
    if(i <= p) cur = min(ret[2 * c - i], p - i);
    while(i - cur - 1 >= 0 && i + cur + 1 < 2*n-1 && aux[i-cur-1] == aux[i+cur+1]){
      cur++;
    }
    ret[i] = cur;
    if(i + ret[i] > p){
      p = i + ret[i];
      c = i;
```

```
    }
  }
}
```

4 Math

4.1 Fraction

```
// dependency: GCD(i64 a, i64 b)
struct frac {
  i64 a, b;
  frac(i64 _a=0, i64 _b=1) : a(_a), b(_b) { if(a == 0 && b == 0) b = 1; assert(b != 0);
  relax(); }
  // Essential: Basic Operations
  void relax() { i64 g = GCD(abs(a), abs(b)); a /= g, b /= g; }
  frac operator + (const frac &ot) const { return { a * ot.b + ot.a * b, b * ot.b }; }
  frac operator - (const frac &ot) const { return { a * ot.b - ot.a * b, b * ot.b }; }
  frac operator * (const frac &ot) const { return { a * ot.a, b * ot.b }; }
  frac operator / (const frac &ot) const { return { a * ot.b, b * ot.a }; }
  frac operator - () { return { -a, b }; }
  // Essential: Basic Comparation
  bool operator == (const frac& ot) const { return a * ot.b == ot.a * b; }
  bool operator < (const frac& ot) const { return a * ot.b < ot.a * b; }
  bool operator <= (const frac& ot) const { return a * ot.b <= ot.a * b; }
  bool operator > (const frac& ot) const { return ot <= *this; }
  bool operator >= (const frac& ot) const { return ot < *this; }
  // Optional: Advanced Operations
  const frac& operator += (const frac &ot) { return *this = *this + ot; }
  const frac& operator -= (const frac &ot) { return *this = *this - ot; }
  const frac& operator *= (const frac &ot) { return *this = *this * ot; }
  const frac& operator /= (const frac &ot) { return *this = *this / ot; }
};
// fraction IO
ostream& operator<< (ostream& os, const frac& frac_x) { return os << frac_x.a << "/" <<
frac_x.b; }
istream& operator>> (istream& os, frac& frac_x) {
  os >> frac_x.a >> frac_x.b;
  frac_x.relax();
  return os;
}
```

4.2 Matrix

```
// Do not use this class as const
typedef i64 ELEM;
const ELEM MOD = 1e9+7; // If don't use MOD, set as 0x7fffffffffffffff
struct mat {
  int n, m;
  vector<vector<ELEM> > ar;
  // ----- constructor, assignment  ----- //
  mat(int n, int m, ELEM x = 0) : n(n), m(m), ar(n, vector<ELEM>(m, x)) {  }
  mat(int n = 0) : mat(n, n) {  }
  mat(const mat& o) { n = o.n, m = o.m, ar = o.ar; }
  mat(const vector<vector<ELEM>>& ar) : n(ar.size()), m(ar.size() ? ar[0].size() : 0),
  ar(ar) { }
  // ----- get field ----- //
  operator const vector<vector<ELEM> >& () const { return ar; }
  vector<ELEM>& operator[](int i) { return ar[i]; }
  const vector<ELEM>& operator[](int i) const { return ar[i]; }
  // ----- calculate ----- //
  mat pow(i64 x) const {
    assert(n == m && 0 <= x);
    mat a(*this), ret = eye(n);
    while(x) {
      if(x%2) ret = ret * a;
      a = a * a;
      x /= 2;
    }
    return ret;
  }
  mat operator * (const mat& o) const {
    assert(m == o.n);
    mat ret(n, o.m);
    FOR(i, 0, n) {
      FOR(j, 0, o.m) {
        FOR(k, 0, m) {
          ret[i][j] += ar[i][k] * o[k][j] % MOD;
          ret[i][j] %= MOD;
        }
      }
    }
    return ret;
  }
  mat operator + (const mat& o) const {
    assert(n == o.n && m == o.m);
    mat ret(n, m);
    FOR(i, 0, n) FOR(j, 0, n) ret[i][j] = (ar[i][j] + o[i][j]) % MOD;
    return ret;
  }
  mat operator - (const mat& o) const {
    assert(n == o.n && m == o.m);
    mat ret(n, m);
    FOR(i, 0, n) FOR(j, 0, n) ret[i][j] = (ar[i][j] - o[i][j]) % MOD;
    return ret;
  }
  mat operator * (const ELEM x) const {
    mat ret = ar;
    FOR(i, 0, n) FOR(j, 0, m) ret[i][j] = ret[i][j] * x % MOD;
    return ret;
  }
  mat operator / (const ELEM x) const {
    mat ret = ar;
    FOR(i, 0, n) FOR(j, 0, m) ret[i][j] = ret[i][j] / x % MOD;
    return ret;
  }
  const mat& operator - () {
    FOR(i, 0, n) FOR(j, 0, m) ar[i][j] = -ar[i][j];
    return *this;
  }
    // If use dp matrix as: state = state * dpmat
    // use rotated dpmat and horizontal state mat
  mat rotate() const {
    mat ret(m, n);
    FOR(i, 0, n) FOR(j, 0, m) ret[j][i] = ar[i][j];
    return ret;
  }
```

```cpp
  static mat eye(const int size) {
    mat ret(size);
    FOR(i, 0, size) ret[i][i] = 1;
    return ret;
  }
  // return matrix dp
    // dp[i] = ar[0] * dp[i-n] + ar[1] * dp[i-n+1] + ... + ar[n-1] * dp[i-1]
  // data matrix br: br[0] = dp[i], br[1] = dp[i-1] ...
  // If DP equation contains constant, fix one element for constant
  static mat dpmat(const vector<ELEM>& ar) {
    int n = ar.size();
    mat ret(n, n);
    FOR(i, 0, n-1) ret[i][i+1] = 1; // transition prev dp values
    FOR(i, 0, n) ret[n-1][i] = ar[i]; // DP equation
    return ret;
  }
};
ostream& operator<<(ostream& os, const mat& v) { for(auto vv : (vector<vector<ELEM> >)v) os
<< vv << ENDL; return os; }
```

4.3 Miller-Rabin Test

```cpp
namespace miller_rabin{ // O(logP)
  i64 mul(i64 x, i64 y, i64 mod){ return (__int128) x * y % mod; }
  i64 ipow(i64 x, i64 y, i64 p){
    i64 ret = 1, piv = x % p;
    while(y){
      if(y&1) ret = mul(ret, piv, p);
      piv = mul(piv, piv, p);
      y >>= 1;
    }
    return ret;
  }
  bool miller_rabin(i64 x, i64 a){
    if(x % a == 0) return 0;
    i64 d = x - 1;
    while(1){
      i64 tmp = ipow(a, d, x);
      if(d&1) return (tmp != 1 && tmp != x-1);
      else if(tmp == x-1) return 0;
      d >>= 1;
    }
  }
  bool isprime(i64 x){
    for(auto &i : {2, 3, 5, 7, 11, 13, 17, 19, 23, 29, 31, 37}){
      if(x == i) return 1;
      if(x > 40 && miller_rabin(x, i)) return 0;
    }
    if(x <= 40) return 0;
    return 1;
  }
}
```

4.4 Euler's Sieve

```cpp
const int RANGE = 2e7;
int pn, spf[RANGE], pr[RANGE]; // spf[x] = min prime factor of x
void eulerSieve() {
  FOR(x, 2, RANGE) {
    if(!spf[x]) spf[x] = pr[pn++] = x; // if x is prime, spf[x] = x
    for(int j = 0; x*pr[j] < RANGE; ++j) {
      spf[x*pr[j]] = pr[j];
      if(x % pr[j] == 0) break;
    }
  }
}
```

4.5 Binomial Coefficient

```cpp
// 1. c[n][r] = c[n-1][r-1] + c[n-1][r]
// 2. nCr = n! / ((n-r)! * r!)
const int RANGE = 1e6;
// precalc: n!, n!^(-1) --> O(NlogP)
void precalc_1() {
  i64 ftr[RANGE], iftr[RANGE];
  ftr[0] = iftr[0] = 1;
  FOR(i, 1, RANGE) {
    ft[i] = (ft[i-1] * (i64)i) % MOD;
    ift[i] = (ift[i-1] * POW((i64)i, MOD-2)) % MOD;
  }
}
// (n-1)!^(-1) =  n*n!^(-1) --> O(n+logP)
void precalc_2() {
  i64 ftr[RANGE], iftr[RANGE];
  ftr[0] = iftr[0] = 1;
  FOR(i, 1, RANGE) ftr[i] = ftr[i-1] * i;
  iftr[RANGE-1] = POW(ftr[RANGE-1], MOD-2);
  RFOR(i, RANGE-2, 0) iftr[i] = (i * iftr[i+1]) % MOD;
}
// inv(1) = 1, inv(1) = -floor(p/i) + inv(p%i) --> O(n)
void precalc_3() {
  i64 inv[RANGE+1], ftr[RANGE], iftr[RANGE];
  inv[1] = 1;
  FOR(i, 2, RANGE+1) {
    inv[i] = inv[MOD % i] - (MOD / i);
    if(inv[i] < 0) inv[i] += MOD;
    inv[i] %= MOD;
  }
}
```

4.6 Inclusion-Exclusion Principle

```cpp
FOR(i, 1, (1 << n)) { // get Union(A, B, C, D ...)
    int bits = __builtin_popcount(i);
    if(bits % 2); // add to ans
    else; // sub to ans
}
```

4.7 Josephus Problem

```cpp
/*  O(n)
  f(n, k) = last survived person for n-people, k-cycle

  < basic idea >
  except 1 element from f(n, k), then answer is f(n-1, k)
  but f(n-1, k) need to be repositioned to starting from kth's next person

  < 1-indexed >
  f(1, k) = 1
  f(n, k) = ((f(n-1, k) + k-1) % n) + 1
```

```
  < 0-indexed >
  f(1, k) = 0;
  f(n, k) = ((f(n-1, k) + k) % n)
*/
// O(KlogN) algorithm
long long joseph (long long n,long long k) {
    if (n==1LL) return 0LL;
    if (k==1LL) return n-1LL;
    if (k>n) return (joseph(n-1LL,k)+k)%n;
    long long cnt=n/k;
    long long res=joseph(n-cnt,k);
    res-=n%k;
    if (res<0LL) res+=n;
    else res+=res/(k-1LL);
    return res;
}
```

5 Segment Tree

5.1 Fenwick Tree Tricks

```
struct fenwick {
  int n;
  vector<i64> t;
  void init(int _n) { n = _n, t = vector<i64>(n+1, 0); }
  void add(int u, i64 x) { for(++u; u < t.size(); u += (u&-u)) t[u] += x; }
  i64 sum(int u) {
        i64 ret = 0;
    for(++u; u > 0; u -= (u&-u)) ret += t[u];
    return ret;
  }
  i64 operator[](int u) {
    ++u;
    int ret = t[u], p = u - (u&-u);
    --u;
    while(u != p) {
      ret -= t[u];
      u -= (u&-u);
    }
    return ret;
  }
  // Can use all elements are positive.
  // k >= 0
  // find x which ( sum[a[0]..a[x-1]] < k <= sum[a[0]..a[x]] )
  int lower(i64 k) {
    if(k < 0) return 0;
    int l = (1 << (8*sizeof(int) - __builtin_clz(n)) - 1);
    int u = 0;
    while(l > 0 && u <= n) {
      int tu = u + l;
      if(k > t[tu]) u = tu, k -= t[tu];
      do l >>= 1;
      while(l > 0 && u + l > n);
    }
    return u;
  }
  // find x which ( sum[a[0]..a[x-1]] <= k < sum[a[0]..a[x]] )
  int upper(i64 k) {
    if(k < 0) return 0;
    int l = (1 << (8*sizeof(int) - __builtin_clz(n)) - 1);
    int u = 0;
    while(l > 0 && u <= n) {
      int tu = u + l;
      if(k >= t[tu]) u = tu, k -= t[tu];
      do l >>= 1;
      while(l > 0 && u + l > n);
    }
    return u;
  }
};
```

5.2 Fenwick Tree 2D (Sparse)

```
struct segtree {
  vi ys[RANGE], t[RANGE];
  // Notify segtree update access on (x, y)
  void initpos(int x, int y) {
    for(++x; x < RANGE; x += (x&-x)) {
      ys[x].pb(y);
    }
  }
  // Execute after notifying (x, y)
  void init() {
    FOR(i, 0, RANGE) sort(ALL(ys[i])), UNIQUE(ys[i]), t[i].assign(ys[i].size()+1, 0);
  }
  // add (x, y) to c
  void add(int x, int y, int c) {
    for(++x; x < RANGE; x += (x&-x)) {
      for(int j = getidx(ys[x], y)+1; j < sz(t[x]); j += (j&-j)) {
        t[x][j] += c;
      }
    }
  }
  // partial sum of ([..x], [..y])
  int sum(int x, int y) {
    int ret = 0;
    for(++x; x > 0; x -= (x&-x)) {
      int j = getidx(ys[x], y);
      if(j == ys[x].size() || ys[x][j] > y) --j;
      for(++j; j > 0; j -= (j&-j)) {
        ret += t[x][j];
      }
    }
    return ret;
  }
} seg;
```

5.3 Fenwick Tree Range Update/Query

```
struct rfenwick { // using 2 basic fenwick tree
  fenwick ax, b;
  void init(int n) { ax.init(n), b.init(n); }
  void add(int u, i64 x) { b.add(u, x); }
  void add(int s, int e, i64 x) {
    ax.add(s, x);
    ax.add(e+1, -x);
    b.add(s, -x * (s-1));
```

```
    b.add(e+1, x * e);
  }
  i64 sum(int u) {
    return u * ax.sum(u) + b.sum(u);
  }
};
```

5.4 Segment Tree (Loop)

```
template<class T, class C>
struct segtree {
  int n;
  vector<T> t;
  void build(const vector<T>& ar) {
    n = ar.size();
    t.assign(n*2, 0);
    FOR(i, 0, n) t[n+i] = ar[i];
    RFOR(i, n-1, 1) t[i] = C()(t[i*2], t[i*2+1]);
  }
  void mod(int u, T k) {
    for(t[u += n] = k; u > 1; u /= 2) t[u/2] = C()(t[u], t[u^1]);
  }
  T query(int s, int e) {
    T ret = 0;
    for(s += n, e += n; s < e; s /= 2, e /= 2) {
      if(s & 1) ret = C()(ret, t[s++]);
      if(e & 1) ret = C()(ret, t[--e]);
    }
    return ret;
  }
};
```

5.5 Segment Tree 2D (Dense)

```
struct segtree {
  int n, m;
  vvi t;
  void init(const vvi& ar) {
    n = ar.size(), m = ar[0].size();
    t.assign(2*n, vi(2*m, 0));
    FOR(y, 0, n) { // push in leaf
      FOR(x, 0, m) {
        t[n+y][m+x] = ar[y][x];
      }
    }
    RFOR(y, 2*n-1, 1) { // construct
      RFOR(x, 2*m-1, 1) {
        if(y < n) t[y][x] = t[y*2][x] + t[y*2+1][x];
        if(x < m) t[y][x] = t[y][x*2] + t[y][x*2+1];
      }
    }
  }
  void modify(int y, int x, int c) {
    t[y + n][x + m] = c; // leaf update
    for(y += n; y > 0; y /= 2) {
      for(int x2 = x + m; x2 > 0; x2 /= 2) {
        if(y < n) t[y][x2] = t[y*2][x2] + t[y*2+1][x2];
        if(x2 < m) t[y][x2] = t[y][x2*2] + t[y][x2*2+1];
      }
    }
  }
  int query(int sy, int sx, int ey, int ex) {
    int ret = 0;
    for(sy += n, ey += n; sy < ey; sy /= 2, ey /= 2) {
      for(int x1 = sx + m, x2 = ex + m; x1 < x2; x1 /= 2, x2 /= 2) {
        if(sy&1) {
          if(x1&1) ret += t[sy][x1];
          if(x2&1) ret += t[sy][x2-1];
        }
        if(ey&1) {
          if(x1&1) ret += t[ey-1][x1];
          if(x2&1) ret += t[ey-1][x2-1];
        }
        if(x1&1) ++x1;
        if(x2&1) --x2;
      }
      if(sy&1) ++sy;
      if(ey&1) --ey;
    }
    return ret;
  }
};
```

5.6 Lazy Propagation

```
const int ST_MAX = 1<<21, lf = ST_MAX/2;
struct segtree{
  i64 t[ST_MAX], d[ST_MAX];
  segtree(){ memset(t, 0, sizeof(t)), memset(d, 0, sizeof(d)); }
  void build(){ RFOR(i, lf-1, 1) t[i] = t[i*2]+ t[i*2+1]; } // !! BUILD !!
  void propagate(int u, int ns, int ne){
    if(!d[u]) return;
    if(u < lf){ // propagate to childs
      d[u*2] += d[u];
      d[u*2+1] += d[u];
    }
    t[u] += d[u] * (ne-ns); // update node
    d[u] = 0;
  }
  void add(int s, int e, int x){ add(s, e, x, 1, 0, lf); } // [s, e)
  void add(int s, int e, int x, int u, int ns, int ne){
    propagate(u, ns, ne);
    if(e <= ns || ne <= s) return;
    if(s <= ns && ne <= e){
      d[u] += x;
      propagate(u, ns, ne);
      return;
    }
    int mid = (ns+ne)/2;
    add(s, e, x, u*2, ns, mid), add(s, e, x, u*2+1, mid, ne);
    t[u] = t[u*2] + t[u*2+1];
  }
  i64 sum(int s, int e){ return sum(s, e, 1, 0, lf); } // [s, e)
  i64 sum(int s, int e, int u, int ns, int ne){
    propagate(u, ns, ne);
    if(e <= ns || ne <= s) return 0;
    if(s <= ns && ne <= e) return t[u];
```

```
    int mid = (ns+ne)/2;
    return sum(s, e, u*2, ns, mid) + sum(s, e, u*2+1, mid, ne);
  }
};
```

5.7 Persistent Segment Tree

```
struct node {
  int x;
  node *l, *r;
  node(int _x = 0, node* l = 0, node* r = 0) : x(_x), l(l), r(r) {  }
  node* addtree(int u, int c, int ns = 0, int ne = MAXN-1) {
    if(ns <= u && u <= ne) {
      if(ns == ne) return new node(x + c, 0, 0);
      int mid = (ns+ne)/2;
      return new node(x + c, l->addtree(u, c, ns, mid), r->addtree(u, c, mid+1, ne));
    }
    return this;
  }
  int query(int s, int e, int ns = 0, int ne = MAXN-1) {
    if(s <= ns && ne <= e) return x;
    if(ne < s || e < ns) return 0;
    int mid = (ns+ne)/2;
    return l->query(s, e, ns, mid) + r->query(s, e, mid+1, ne);
  }
} *root[MAXN+1];
```

5.8 Persistent Segment Tree (Array)

```
struct pst {
  i64 x[MAXN*LOGN];
  int l[MAXN*LOGN], r[MAXN*LOGN], tcnt;
  int base(int ns = 0, int ne = MAXN-1) { // make 0th tree
    int u = tcnt++;
    l[u] = u, r[u] = u;
  }
  int make(int idx, int c, int u, int ns = 0, int ne = MAXN-1) { // update from u-rooted
    if(idx < ns || ne < idx) return u;
    int v = tcnt++;
    if(ns == ne) x[v] = (x[u] + c) % MOD;
    else {
      int m = (ns+ne)/2;
      l[v] = make(idx, c, l[u], ns, m);
      r[v] = make(idx, c, r[u], m+1, ne);
      x[v] = (x[l[v]] + x[r[v]]) % MOD;
    }
    return v;
  }
  i64 query(int s, int e, int u, int ns = 0, int ne = MAXN-1) { // query from u-rooted
    if(s <= ns && ne <= e) return x[u];
    if(ne < s || e < ns) return 0;
    int m = (ns+ne)/2;
    return (query(s, e, l[u], ns, m) + query(s, e, r[u], m+1, ne)) % MOD;
  }
};
```

5.9 HLD (Vertex)

```
/*
  HLD with costed vertex.
  usually (dfs_init, lca, decomposite, eidx, query) don't need to be changed.
  just modify (segtree, init_segs), and if segtree function changed modify (update,
query_to)
*/
const int MAXN = 1e5+10, LOGN = 18, INF = 0x7fffffff;
struct hld_vtx {
  struct segtree {
    int n;
    vi t;
    void init(int _n) { n = _n; t.assign(2*n, INF); }
    void update(int u, int x) {
      for(t[u += n] = x; u > 1; u /= 2) t[u/2] = min(t[u], t[u^1]);
    }
    int query(int s, int e) {
      int ret = INF;
      for(s += n, e += n; s < e; s /= 2, e /= 2) {
        if(s&1) ret = min(ret, t[s++]);
        if(e&1) ret = min(ret, t[--e]);
      }
      return ret;
    }
  };
  int n, rt;
  vi ssz, dep, hidx;
  vvi g, par, hvy;
  vector<segtree> segs;
  hld_vtx(vvi& g, int rt) : g(g), rt(rt), n(g.size()), ssz(n, 0), dep(n, 0), hidx(n, -1),
  par(n, vi(LOGN, 0)) {
    par[rt][0] = rt;
    dfs_init(rt);
    decomposite(rt);
    init_segs();
  }
  void dfs_init(int u) { // initialize dfs info
    ssz[u] = 1;
    FOR(j, 1, LOGN) par[u][j] = par[par[u][j-1]][j-1];
    for(int v : g[u]) {
      if(par[u][0] == v) continue;
      par[v][0] = u;
      dep[v] = dep[u] + 1;
      dfs_init(v);
      ssz[u] += ssz[v];
    }
  }
  int lca(int u, int v) { // consider par[root] = root
    if(dep[u] < dep[v]) swap(u, v);
    int dif = dep[u] - dep[v];
    FOR(j, 0, LOGN) if(dif & (1<<j)) u = par[u][j];
    if(u != v) {
      RFOR(j, LOGN-1, 0) if(par[u][j] != par[v][j]) u = par[u][j], v = par[v][j];
      u = par[u][0];
    }
    return u;
  }
  void decomposite(int rt) { // decomposite tree
    queue<int> q;
```

```
    q.push(rt);
    while(q.size()) {
      int u = q.front(); q.pop();
      for(int v : g[u]) if(par[v][0] == u) q.push(v);
      int p = par[u][0];
      if(u != rt && ssz[u]*2 >= ssz[p]) { // extend h-path
        hidx[u] = hidx[p];
        hvy[hidx[u]].pb(u);
      } else { // create h-path
        hidx[u] = hvy.size();
        hvy.pb(vi());
        hvy[hidx[u]].pb(u);
      }
    }
  }
  void init_segs() { // initialize segtrees
    segs.assign(hvy.size(), segtree());
    FOR(i, 0, hvy.size()) segs[i].init(hvy[i].size()); // m nodes
  }
  int vidx(int u) { // get v's index in h-path
    return dep[u] - dep[hvy[hidx[u]][0]];
  }
  void update(int u, int x) { // update v's cost
    if(x == 0) segs[hidx[u]].update(vidx(u), INF);
    else segs[hidx[u]].update(vidx(u), vidx(u));
  }
  int query(int v) { // root->v query
    return query_to(0, v);
  }
  int query_to(int u, int v) { // return u->v path's query
    if(hidx[u] == hidx[v]) {
      int res = segs[hidx[u]].query(vidx(u), vidx(v)+1);
      if(res == INF) return INF;
      return hvy[hidx[u]][res];
    }
    int res = query_to(u, par[hvy[hidx[v]][0]][0]);
    if(res != INF) return res;
    res = segs[hidx[v]].query(0, vidx(v)+1);
    if(res == INF) return INF;
    return hvy[hidx[v]][res];
  }
};
```

5.10 HLD (Edge)

```
/*
  HLD with costed edge.
  Unlike the normal HLD, top edge of each chains are also belongs to chain.

  usually (dfs_init, lca, decomposite, eidx, query) don't need to be changed.
  just modify (segtree, init_segs), and if segtree function changed modify (update,
query_to)
*/
const int LOGN = 18;
struct hld_edge {
  struct segtree { // just edit segtree ( currently half-open interval [s, e) )
    int n;
    vi t;
    void init(int _n) { n = _n; t.assign(2*n, 0); }
    void update(int u, int x) {
      for(t[u += n] = x; u > 1; u /= 2) t[u/2] = max(t[u], t[u^1]);
    }
    int query(int s, int e) {
      int ret = 0;
      for(s += n, e += n; s < e; s /= 2, e /= 2) {
        if(s&1) ret = max(ret, t[s++]);
        if(e&1) ret = max(ret, t[--e]);
      }
      return ret;
    }
  };
  int n, rt;
  vi ssz, dep, hidx;
  vvi g, par, hvy;
  vector<segtree> segs;
  hld_edge(vvi& g, int rt) : g(g), rt(rt), n(g.size()), ssz(n, 0), dep(n, 0), hidx(n, -1),
  par(n, vi(LOGN, 0)) {
    par[rt][0] = rt;
    dfs_init(rt);
    decomposite(rt);
    init_segs();
  }
  void dfs_init(int u) { // initialize dfs info
    ssz[u] = 1;
    FOR(j, 1, LOGN) par[u][j] = par[par[u][j-1]][j-1];
    for(int v : g[u]) {
      if(par[u][0] == v) continue;
      par[v][0] = u;
      dep[v] = dep[u] + 1;
      dfs_init(v);
      ssz[u] += ssz[v];
    }
  }
  int lca(int u, int v) { // consider par[root] = root
    if(dep[u] < dep[v]) swap(u, v);
    int dif = dep[u] - dep[v];
    FOR(j, 0, LOGN) if(dif & (1<<j)) u = par[u][j];
    if(u != v) {
      RFOR(j, LOGN-1, 0) if(par[u][j] != par[v][j]) u = par[u][j], v = par[v][j];
      u = par[u][0];
    }
    return u;
  }
  void decomposite(int rt) { // decomposite tree
    hidx[rt] = -1;
    queue<int> q;
    q.push(rt);
    while(q.size()) {
      int u = q.front(); q.pop();
      for(int v : g[u]) if(par[v][0] == u) q.push(v);
      if(u != rt) {
        int p = par[u][0];
        if(p != rt && ssz[u]*2 >= ssz[p]) { // extend h-path (only if h-path)
          hidx[u] = hidx[p];
```

```
        hvy[hidx[u]].pb(u);
      } else { // create h-path (l-path or root-h-path)
        hidx[u] = hvy.size();
        hvy.pb(vi());
        hvy[hidx[u]].pb(p);
        hvy[hidx[u]].pb(u);
      }
    }
  }
}
void init_segs() { // initialize segtrees
  segs.assign(hvy.size(), segtree());
  FOR(i, 0, hvy.size()) segs[i].init(hvy[i].size()-1); // m vertices: m-1 edges
}
int eidx(int v) { // get u->v edge index in h-path
  return dep[par[v][0]] - dep[hvy[hidx[v]][0]];
}
void update(int u, int v, int x) { // u->v edge update
  if(par[u][0] == v) swap(u, v);
  assert(par[v][0] == u);
  segs[hidx[v]].update(eidx(v), x);
}
int query_to(int u, int v) { // return u->v path's query
  if(u == v) return 0;
  // modify range if segtree use closed interval [s, e]
  if(hidx[u] == hidx[v]) return segs[hidx[u]].query(eidx(u)+1, eidx(v)+1); // e(u)+1
  because target is edge
  return max(query_to(u, hvy[hidx[v]][0]), segs[hidx[v]].query(0, eidx(v)+1)); // query
  tail path + recur
}
int query(int u, int v) {
  int t = lca(u, v);
  return max(query_to(t, u), query_to(t, v));
}
};
```

6 Miscellaneous

6.1 Preset

```
#include <bits/stdc++.h>
#include <ext/pb_ds/assoc_container.hpp>
#include <ext/pb_ds/tree_policy.hpp>
using namespace std;
using namespace __gnu_pbds;
template<class key, class value, class cmp = less<key>>
using treemap = tree<key, value, less<int>, rb_tree_tag, tree_order_statistics_node_update>;
// key, val, comp, implements, 노드 불변 규칙
template<class key, class cmp = less<key>>
using treeset = tree<key, null_type, cmp, rb_tree_tag, tree_order_statistics_node_update>;
#ifdef LOCAL_BOOKNU
#define debug(...) cerr << "[" << #__VA_ARGS__ << "]:", debug_out(__VA_ARGS__)
#else
#define debug(...) 42
#endif
#define FOR(i, f, n) for(int (i) = (f); (i) < (int)(n); ++(i))
#define RFOR(i, f, n) for(int (i) = (f); (i) >= (int)(n); --(i))
#define pb push_back
#define emb emplace_back
#define fi first
#define se second
#define ENDL '\n'
#define sz(A) (int)(A).size()
#define ALL(A) A.begin(), A.end()
#define UNIQUE(c) (c).resize(unique(ALL(c)) - (c).begin())
typedef pair<int, int> ii;
typedef pair<int, ii> iii;
typedef vector<int> vi;
typedef vector<vi> vvi;
typedef vector<ii> vii;
typedef vector<vii> vvii;
typedef long long i64;
typedef unsigned long long ui64;
inline int getidx(const vi& ar, int x) { return lower_bound(ALL(ar), x) - ar.begin(); }
inline i64 GCD(i64 a, i64 b) { i64 n; if(a < b) swap(a, b); while(b != 0) { n = a % b; a =
b; b = n; } return a; }
inline i64 LCM(i64 a, i64 b) { if(a == 0 || b == 0) return GCD(a, b); return a / GCD(a, b) *
b; }
inline i64 CEIL(i64 n, i64 d) { return n / d + (i64)(n % d != 0); } // for positive numbers
inline i64 ROUND(i64 n, i64 d) { return n / d + (i64)((n % d) * 2 >= d); }
const i64 MOD = 1e9+7;
inline i64 POW(i64 a, i64 n) {
  i64 ret;
  for(ret = 1; n; a = a*a%MOD, n /= 2) { if(n%2) ret = ret*a%MOD; }
  return ret;
}
template <class T> ostream& operator<<(ostream& os, vector<T> v) {
  os << "[";
  int cnt = 0;
  for(auto vv : v) { os << vv; if(++cnt < v.size()) os << ","; }
  return os << "]";
}
template <class T> ostream& operator<<(ostream& os, set<T> v) {
  os << "[";
  int cnt = 0;
  for(auto vv : v) { os << vv; if(++cnt < v.size()) os << ","; }
  return os << "]";
}
template <class L, class R> ostream& operator<<(ostream& os, pair<L, R> p) { return os <<
"(" << p.fi << "," << p.se << ")"; }
void debug_out() { cerr << endl; }
template <typename Head, typename... Tail> void debug_out(Head H, Tail... T) { cerr << " "
<< H, debug_out(T...); }
// .......................... MAIN .......................... //
void input() { }
int solve() { return 0; }
void execute() { input(), solve(); }
int main(void) {
#ifdef LOCAL_BOOKNU
  freopen("__IO/input.txt", "r", stdin);
  // freopen("__IO/out.txt", "w", stdout);
#endif
  cin.tie(0), ios_base::sync_with_stdio(false);
  execute();
```

```
  return 0;
}
```

6.2 3D-Partial Sum

```
const int RANGE = 256;
int n, k, ps[RANGE][RANGE][RANGE], ar[RANGE][RANGE][RANGE];
int f(int x, int y, int z) {
  return (x < 0 || y < 0 || z < 0 || x >= RANGE || y >= RANGE || z >= RANGE ? 0 :
  ps[x][y][z]);
}
int sum(int x1, int y1, int z1, int l) {
  int x2 = min(RANGE - 1, x1 + l), y2 = min(RANGE - 1, y1 + l), z2 = min(RANGE - 1, z1 + l);
  --x1, --y1, --z1;
  return
    f(x2, y2, z2)
    - f(x1, y2, z2) - f(x2, y1, z2) - f(x2, y2, z1)
    + f(x1, y1, z2) + f(x1, y2, z1) + f(x2, y1, z1)
    - f(x1, y1, z1);
}
void init() {
  FOR(x, 0, RANGE) {
    FOR(y, 0, RANGE) {
      FOR(z, 0, RANGE) {
        ps[x][y][z] +=
          f(x - 1, y, z) + f(x, y - 1, z) + f(x, y, z - 1)
          - f(x - 1, y - 1, z) - f(x - 1, y, z - 1) - f(x, y - 1, z - 1)
          + f(x - 1, y - 1, z - 1);
      }
    }
  }
}
```

6.3 Knuth's Optimization

If three conditions satisfies in DP equation, the time complexity can be reduced from $O(n^3)$ to $O(n^2)$

1. DP Equation Form
   - $D[i][j] = \min_{i<k<j}(D[i][k] + D[k][j]) + C[i][j]$
2. Quadrangle Inequality
   - $C[a][c] + C[b][d] \leq C[a][d] + C[b][c], a \leq b \leq c \leq d$
3. Monotonicity
   - $C[b][c] \leq C[a][d], a \leq b \leq c \leq d$

Define $A[i][j] = k$ for $D[i][j]$ becomes minimum
If condition 2, 3 been satisfied, bellowing inequality holds.
$A[i][j-1] \leq A[i][j] \leq A[i+1][j]$

6.4 Bit Tricks

```
__builtin_clz(int x); // count leading-zero
__builtin_ctz(int x); // count tailing-zero
__builtin_clzll(i64 x);
__builtin_ctzll(i64 x);
__builtin_popcount(int x); // number of 1-bits
__builtin__ffs(int x); // lsb index (1-based, x = 0 -> 0)
floor(log2(n)): 31 - __builtin_clz(n|1);
// 00111, 01011, 01101, 01110, 10011, 10101...
i64 next_perm(i64 x) {
    i64 t = x|(x-1);
    return (t + 1) | (((~t & -~t) - 1) >> (__builtin_ctz(x) + 1))
}
```

6.5 Fast IO

```
class FastIO {
  int fd, bufsz;
  char *buf, *cur, *end;
public:
  FastIO(int _fd = 0, int _bufsz = 1 << 20) : fd(_fd), bufsz(_bufsz) {
    buf = cur = end = new char[bufsz];
  }
  ~FastIO() { delete[] buf; }
  bool readbuf() {
    cur = buf;
    end = buf + bufsz;
    while(true) {
      size_t res = fread(cur, sizeof(char), end - cur, stdin);
      if(res == 0) break;
      cur += res;
    }
    end = cur;
    cur = buf;
    return buf != end;
  }
  bool hasNext() {
    while(true) {
      if(cur == end && !readbuf()) return false;
      if(isdigit(*cur) || *cur == '-') break;
      ++cur;
    }
    return true;
  }
  int r() {
    while(true) {
      if(cur == end) readbuf();
      if(isdigit(*cur) || *cur == '-') break;
      ++cur;
    }
    bool sign = true;
    if(*cur == '-') {
      sign = false;
      ++cur;
    }
    int ret = 0;
    while(true) {
      if(cur == end && !readbuf()) break;
      if(!isdigit(*cur)) break;
      ret = ret * 10 + (*cur - '0');
      ++cur;
    }
    return sign ? ret : -ret;
  }
} sc;
```

6.6 Input Format

```
while(scanf("%d", &n) > 0) { // until input ends
    scanf("%d: (%d)", &x, &y); // formatted input
    for(int i = 0; i < x; ++i) scanf("%d", &z);
}
getline(cin, line);
```

6.7 String Parsing

```
vector<string> split(string& target, string regex) { // using regex
  vector<string> ret;
  std::regex rgx(regex);
  std::sregex_token_iterator iter(target.begin(),
    target.end(),
    rgx,
    -1);
  std::sregex_token_iterator end;
  for( ; iter != end; ++iter) ret.pb(*iter);
  return ret;
}
vector<string> space_split(string& s) { // split by whitespace
  istringstream iss(s);
  vector<string> ret(istream_iterator<string>{iss}, istream_iterator<string>());
  return ret;
}
std::to_string(42);
std::atoi("42");
```

6.8 Ordered Statistics Tree in g++

```
s.find_by_order(x); // 0-indexed
s.order_of_key(x) // 0-indexed, find first element x <= ar[idx]
```

6.9 Prime Numbers

```
  < 10^k          number     divisors   2 3 5 71113171923293137
  ---------------------------------------------------------------
  1                    6            4   1 1
  2                   60           12   2 1 1
  3                  840           32   3 1 1 1
  4                 7560           64   3 3 1 1
  5                83160          128   3 3 1 1 1
  6               720720          240   4 2 1 1 1 1
  7              8648640          448   6 3 1 1 1 1
  8             73513440          768   5 3 1 1 1 1 1
  9            735134400         1344   6 3 2 1 1 1 1
  10          6983776800         2304   5 3 2 1 1 1 1 1
  11         97772875200         4032   6 3 2 2 1 1 1 1
  12        963761198400         6720   6 4 2 1 1 1 1 1 1
  13       9316358251200        10752   6 3 2 1 1 1 1 1 1 1
  14      97821761637600        17280   5 4 2 2 1 1 1 1 1 1
  15     866421317361600        26880   6 4 2 1 1 1 1 1 1 1 1
  16    8086598962041600        41472   8 3 2 2 1 1 1 1 1 1 1
  17   74801040398884800        64512   6 3 2 2 1 1 1 1 1 1 1 1
  18  897612484786617600       103680   8 4 2 2 1 1 1 1 1 1 1 1

  < 10^k    prime   # of prime          < 10^k              prime
  ---------------------------------------------------------------
  1             7            4          10             9999999967
  2            97           25          11            99999999977
  3           997          168          12           999999999989
  4          9973         1229          13          9999999999971
  5         99991         9592          14         99999999999973
  6        999983        78498          15        999999999999989
  7       9999991       664579          16       9999999999999937
  8      99999989      5761455          17      99999999999999997
  9     999999937     50847534          18     999999999999999989
```

6.10 Random

```
mt19937 rng(chrono::steady_clock::now().time_since_epoch().count());
mt19937 rng(0x14004); // 0x94949
int randint(int s, int e) { return uniform_int_distribution<int>(s, e)(rng); }
```

6.11 Hashing

```
struct chash {
  const int RANDOM = (long long)(make_unique<char>().get()) ^
  chrono::high_resolution_clock::now().time_since_epoch().count();
  static unsigned long long hash_f(unsigned long long x) {
    x += 0x9e3779b97f4a7c15;
    x = (x ^ (x >> 30)) * 0xbf58476d1ce4e5b9;
    x = (x ^ (x >> 27)) * 0x94d049bb133111eb;
    return x ^ (x >> 31);
  }
  static unsigned hash_combine(unsigned a, unsigned b) { return a * 31 + b; }
  int operator()(int x) const { return hash_f(x)^RANDOM; }
};

gp_hash_table<key, int, chash> mp;
int main() {
  mp[1] = 1;
}
```