AdMU Progvar

Team Notebook

02/11/2019

Contents

1. Code Templates

```
#include <bits/stdc++.h>
typedef long long ll;
typedef unsigned long long ull;
typedef std::pair<int, int> ii;
typedef std::vector<int> vi;
typedef std::vector<vi> vvi;
typedef std::vector<ii> vii;
const int INF = ~(1<<31);
const ll LINF = (1LL << 60);
const double EPS = 1e-9;
const double pi = acos(-1);
```

2. Data Structures

2.1. **Fenwick Tree.**

2.1.1. *Fenwick Tree w/ Point Queries.*

```
struct fenwick {
- vi ar;
- fenwick(vi &_ar) : ar(_ar.size(), 0) {
--- for (int i = 0; i < ar.size(); ++i) {
----- ar[i] += _ar[i];
----- int j = i | (i+1);
----- if (j < ar.size())
------- ar[j] += ar[i];
--- }
- }
- int sum(int i) {
--- int res = 0;
--- for (; i >= 0; i = (i & (i+1)) - 1)
----- res += ar[i];
--- return res;
- }
- int sum(int i, int j) {  return sum(j) - sum(i-1);  }
- void add(int i, int val) {
--- for (; i < ar.size(); i |= i+1)
----- ar[i] += val;
- }
- int get(int i) {
--- int res = ar[i];
--- if (i) {
----- int lca = (i & (i+1)) - 1;
----- for (--i; i != lca; i = (i&(i+1))-1)
------- res -= ar[i];
--- }
--- return res;
- }
- void set(int i, int val) {  add(i, -get(i) + val);  }
- // range update, point query //
- void add(int i, int j, int val) {
--- add(i, val);
--- add(j+1, -val);
- }
- int get1(int i) { return sum(i); }
- //////////////////////////////
};
```

2.1.2. *Fenwick Tree w/ Max Queries.*

```
struct fenwick {
- vi ar;
- fenwick(vi &_ar) : ar(_ar.size(), 0) {
--- for (int i = 0; i < ar.size(); ++i) {
----- ar[i] += _ar[i];
----- int j = i | (i+1);
----- if (j < ar.size())
------- ar[j] += ar[i];
--- }
- }
- void set(int i, int v) {
--- for (; i < ar.size(); i |= i+1)
----- ar[i] = std::max(ar[i], v);
- }
- // max[0..i]
- int max(int i) {
--- int res = -INF;
--- for (; i >= 0; i = (i & (i+1)) - 1)
----- res = std::max(res, ar[i]);
--- return res;
- }
};
```

2.2. **Segment Tree.**

2.2.1. *Recursive Segment Tree (Point-update).*

```
struct segtree { ----------------------------------------//0ca
- int i, j, val; ----------------------------------------//0cb
- segtree *l, *r; ---------------------------------------//0cc
- segtree(int *ar, int _i, int _j) : i(_i), j(_j) { ------//0cd
--- if (i == j) { ---------------------------------------//0ce
----- val = ar[i]; --------------------------------------//0cf
----- l = r = NULL; -------------------------------------//0d0
--- } else { --------------------------------------------//0d1
----- int k = (i+j) >> 1; -------------------------------//0d2
----- l = new segtree(ar, i, k); ------------------------//0d3
----- r = new segtree(ar, k+1, j); ----------------------//0d4
----- val = l->val + r->val; ----------------------------//0d5
--- } ---------------------------------------------------//0d6
- } -----------------------------------------------------//0d7
- void update(int _i, int _val) { -----------------------//0d8
--- if (i == _i and _i == j) { --------------------------//0d9
----- val = _val; ---------------------------------------//0da
--- } else if (_i < i or j < _i) { ----------------------//0db
----- // do nothing ------------------------------------//0dc
--- } else { --------------------------------------------//0dd
----- l->update(_i, _val); ------------------------------//0de
----- r->update(_i, _val); ------------------------------//0df
----- val = l->val + r->val; ----------------------------//0e0
--- } ---------------------------------------------------//0e1
- } -----------------------------------------------------//0e2
- int query(int _i, int _j) { ---------------------------//0e3
--- if (_i <= i and j <= _j) { --------------------------//0e4
----- return val; ---------------------------------------//0e5
--- } else if (_j < i or j < _i) { ----------------------//0e6
----- return 0; -----------------------------------------//0e7
--- } else { --------------------------------------------//0e8
----- return l->query(_i, _j) + r->query(_i, _j); -------//0e9
--- } ---------------------------------------------------//0ea
- } -----------------------------------------------------//0eb
}; ------------------------------------------------------//0ec
```

2.2.2. *Iterative Segment Tree (Point-update and operation can be non-commutative).*

```
struct segtree { ----------------------------------------//052
- int n; ------------------------------------------------//053
- int *vals; --------------------------------------------//054
- segtree(int *ar, int n) { -----------------------------//055
--- this->n = n; ----------------------------------------//056
--- vals = new int[2*n]; --------------------------------//057
--- for (int i = 0; i < n; ++i) -------------------------//058
----- vals[i+n] = ar[i]; --------------------------------//059
--- for (int i = n-1; i > 0; --i) -----------------------//05a
----- vals[i] = vals[i<<1] + vals[i<<1|1]; --------------//05b
- } -----------------------------------------------------//05c
- void update(int i, int v) { ---------------------------//05d
--- for (vals[i += n] = v; i > 1; i >>= 1) --------------//05e
----- vals[i>>1] = vals[i] + vals[i^1]; -----------------//05f
- } -----------------------------------------------------//060
- int query(int l, int r) { -----------------------------//061
--- int res = 0; ----------------------------------------//062
--- for (l += n, r += n+1; l < r; l >>= 1, r >>= 1) { ----//063
----- if (l&1)  res += vals[l++]; -----------------------//064
----- if (r&1)  res += vals[--r]; -----------------------//065
--- } ---------------------------------------------------//066
--- return res; -----------------------------------------//067
- } -----------------------------------------------------//068
}; ------------------------------------------------------//069
```

2.2.3. *Lazy Segment Tree (Range-update).*

```
struct segtree { ----------------------------------------//06a
- int i, j, val, temp_val = 0; --------------------------//06b
- segtree *l, *r; ---------------------------------------//06c
- segtree(int *ar, int _i, int _j) : i(_i), j(_j) { ------//06d
--- if (i == j) { ---------------------------------------//06e
----- val = ar[i]; --------------------------------------//06f
----- l = r = NULL; -------------------------------------//070
--- } else { --------------------------------------------//071
----- int k = (i + j) >> 1; -----------------------------//072
----- l = new segtree(ar, i, k); ------------------------//073
----- r = new segtree(ar, k+1, j); ----------------------//074
----- val = l->val + r->val; ----------------------------//075
--- } ---------------------------------------------------//076
- } -----------------------------------------------------//077
- void visit() { ----------------------------------------//078
--- if (temp_val) { -------------------------------------//079
----- val += (j-i+1) * temp_val; ------------------------//07a
----- if (l) { ------------------------------------------//07b
------- l->temp_val += temp_val; ------------------------//07c
------- r->temp_val += temp_val; ------------------------//07d
----- } -------------------------------------------------//07e
----- temp_val = 0; -------------------------------------//07f
--- } ---------------------------------------------------//080
- } -----------------------------------------------------//081
- void increase(int _i, int _j, int _inc) { -------------//082
--- visit(); --------------------------------------------//083
--- if (_i <= i && j <= _j) { ---------------------------//084
----- temp_val += _inc; ---------------------------------//085
----- visit(); ------------------------------------------//086
--- } else if (_j < i or j < _i) { ----------------------//087
----- // do nothing ------------------------------------//088
--- } else { --------------------------------------------//089
----- l->increase(_i, _j, _inc); ------------------------//08a
----- r->increase(_i, _j, _inc); ------------------------//08b
----- val = l->val + r->val; ----------------------------//08c
--- } ---------------------------------------------------//08d
- } -----------------------------------------------------//08e
- int query(int _i, int _j) { ---------------------------//08f
--- visit(); --------------------------------------------//090
--- if (_i <= i and j <= _j) { --------------------------//091
----- return val; ---------------------------------------//092
--- } else if (_j < i || j < _i) { ----------------------//093
----- return 0; -----------------------------------------//094
--- } else { --------------------------------------------//095
----- return l->query(_i, _j) + r->query(_i, _j); -------//096
--- } ---------------------------------------------------//097
- } -----------------------------------------------------//098
}; ------------------------------------------------------//099
```

2.2.4. *Persistent Segmentr Tree (Point-update).*

```
struct node {  int l, r, lid, rid, val;  }; -------------//09a
struct segtree { ----------------------------------------//09b
- node *nodes; ------------------------------------------//09c
- int n, node_cnt = 0; ----------------------------------//09d
- segtree(int n, int capacity) { ------------------------//09e
--- this->n = n; ----------------------------------------//09f
--- nodes = new node[capacity]; -------------------------//0a0
- } -----------------------------------------------------//0a1
- int build (int *ar, int l, int r) { -------------------//0a2
--- if (l > r)  return -1; ------------------------------//0a3
--- int id = node_cnt++; --------------------------------//0a4
--- nodes[id].l = l; ------------------------------------//0a5
--- nodes[id].r = r; ------------------------------------//0a6
--- if (l == r) { ---------------------------------------//0a7
----- nodes[id].lid = -1; -------------------------------//0a8
----- nodes[id].rid = -1; -------------------------------//0a9
----- nodes[id].val = ar[l]; ----------------------------//0aa
--- } else { --------------------------------------------//0ab
----- int m = (l + r) / 2; ------------------------------//0ac
----- nodes[id].lid = build(ar, l, m); ------------------//0ad
----- nodes[id].rid = build(ar, m+1, r); ----------------//0ae
----- nodes[id].val = nodes[nodes[id].lid].val + --------//0af
------------------- nodes[nodes[id].rid].val; -----------//0b0
--- } ---------------------------------------------------//0b1
--- return id; ------------------------------------------//0b2
- } -----------------------------------------------------//0b3
- int update(int id, int idx, int delta) { --------------//0b4
--- if (id == -1) ---------------------------------------//0b5
----- return -1; ----------------------------------------//0b6
--- if (idx < nodes[id].l or nodes[id].r < idx) ---------//0b7
----- return id; ----------------------------------------//0b8
--- int nid = node_cnt++; -------------------------------//0b9
--- nodes[nid].l = nodes[id].l; -------------------------//0ba
--- nodes[nid].r = nodes[id].r; -------------------------//0bb
--- nodes[nid].lid = update(nodes[id].lid, idx, delta); --//0bc
--- nodes[nid].rid = update(nodes[id].rid, idx, delta); --//0bd
--- nodes[nid].val = nodes[id].val + delta; -------------//0be
--- return nid; -----------------------------------------//0bf
- } -----------------------------------------------------//0c0
- int query(int id, int l, int r) { ---------------------//0c1
--- if (r < nodes[id].l or nodes[id].r < l) -------------//0c2
----- return 0; -----------------------------------------//0c3
--- if (l <= nodes[id].l and nodes[id].r <= r) ----------//0c4
----- return nodes[id].val; -----------------------------//0c5
--- return query(nodes[id].lid, l, r) + -----------------//0c6
--------- query(nodes[id].rid, l, r); -------------------//0c7
-- } -----------------------------------------------------//0c8
 }; -----------------------------------------------------//0c9
```

2.3. **Sparse Table.**

2.3.1. *1D Sparse Table.*

```
int lg[MAXN+1], spt[20][MAXN]; ---------------------------//0ed
void build(int arr[], int n) { ---------------------------//0ee
- for (int i = 2; i <= n; ++i) lg[i] = lg[i>>1] + 1; -----//0ef
- for (int i = 0; i < n; ++i) spt[0][i] = arr[i]; --------//0f0
- for (int j = 0; (2 << j) <= n; ++j) --------------------//0f1
--- for (int i = 0; i + (2 << j) <= n; ++i) --------------//0f2
----- spt[j+1][i] = min(spt[j][i], spt[j][i+(1<<j)]); ----//0f3
} --------------------------------------------------------//0f4
int query(int a, int b) { --------------------------------//0f5
- int k = lg[b-a+1], ab = b - (1<<k) + 1; ----------------//0f6
- return min(spt[k][a], spt[k][ab]); ---------------------//0f7
} --------------------------------------------------------//0f8
```

2.3.2. *2D Sparse Table.*

```
int lg[N], A[N][N], st[LGN][LGN][N][N]; ------------------//0f9
void build(int n, int m) { -------------------------------//0fa
- for(int k = 2; k <= max(n,m); ++k) lg[k] = lg[k>>1] + 1;//0fb
- for(int i = 0; i < n; ++i) -----------------------------//0fc
--- for(int j = 0; j < m; ++j) ---------------------------//0fd
----- st[0][0][i][j] = A[i][j]; --------------------------//0fe
- for(int bj = 0; (2 << bj) <= m; ++bj) ------------------//0ff
--- for(int j = 0; j + (2 << bj) <= m; ++j) --------------//100
----- for(int i = 0; i < n; ++i) -------------------------//101
------- st[0][bj+1][i][j] = max(st[0][bj][i][j], ---------//102
------------------------------ st[0][bj][i][j + (1 << bj)]);
- for(int bi = 0; (2 << bi) <= n; ++bi) ------------------//104
--- for(int i = 0; i + (2 << bi) <= n; ++i) --------------//105
----- for(int j = 0; j < m; ++j) -------------------------//106
------- st[bi+1][0][i][j] = max(st[bi][0][i][j], ---------//107
------------------------------ st[bi][0][i + (1 << bi)][j]);
- for(int bi = 0; (2 << bi) <= n; ++bi) ------------------//109
--- for(int i = 0; i + (2 << bi) <= n; ++i) --------------//10a
----- for(int bj = 0; (2 << bj) <= m; ++bj) --------------//10b
------- for(int j = 0; j + (2 << bj) <= m; ++j) { --------//10c
--------- int ik = i + (1 << bi); ------------------------//10d
--------- int jk = j + (1 << bj); ------------------------//10e
--------- st[bi+1][bj+1][i][j] = -------------------------//10f
----------- max(max(st[bi][bj][i][j], max(st[bi][bj][ik][j], -//110
----------- max(st[bi][bj][i][jk],st[bi][bj][ik][jk]))); -//111
------- } ------------------------------------------------//112
} --------------------------------------------------------//113
int query(int x1, int x2, int y1, int y2) { --------------//114
- int kx = lg[x2 - x1 + 1],   ky = lg[y2 - y1 + 1]; ------//115
- int x12 = x2 - (1<<kx) + 1, y12 = y2 - (1<<ky) + 1; ----//116
- return max(max(st[kx][ky][x1][y1],  st[kx][ky][x1][y12]),
------------ max(st[kx][ky][x12][y1], st[kx][ky][x12][y12]));
} --------------------------------------------------------//119
```

2.4. **Sqrt Decomposition.**

2.5. **Treap.**

2.5.1. *Explicit Treap.*

2.5.2. *Implicit Treap.*

```
struct cartree { -----------------------------------------//11a
- typedef struct _Node { ---------------------------------//11b
--- int node_val, subtree_val, delta, prio, size; --------//11c
--- _Node *l, *r; ----------------------------------------//11d
--- _Node(int val) : node_val(val), subtree_val(val), ----//11e
------- delta(0), prio((rand()<<16)^rand()), size(1), ----//11f
------- l(NULL), r(NULL) {} ------------------------------//120
--- ~_Node() { delete l; delete r; } ---------------------//121
- } *Node; -----------------------------------------------//122
- int get_subtree_val(Node v) { --------------------------//123
--- return v ? v->subtree_val : 0;  } --------------------//124
- int get_size(Node v) { return v ? v->size : 0; } -------//125
- void apply_delta(Node v, int delta) { ------------------//126
--- if (!v) return; --------------------------------------//127
--- v->delta += delta; -----------------------------------//128
--- v->node_val += delta; --------------------------------//129
--- v->subtree_val += delta * get_size(v); ---------------//12a
- } ------------------------------------------------------//12b
- void push_delta(Node v) { ------------------------------//12c
--- if (!v) return; --------------------------------------//12d
--- apply_delta(v->l, v->delta); -------------------------//12e
--- apply_delta(v->r, v->delta); -------------------------//12f
--- v->delta = 0; ----------------------------------------//130
- } ------------------------------------------------------//131
- void update(Node v) { ----------------------------------//132
--- if (!v) return; --------------------------------------//133
--- v->subtree_val = get_subtree_val(v->l) + v->node_val -//134
------------------ + get_subtree_val(v->r); --------------//135
--- v->size = get_size(v->l) + 1 + get_size(v->r); -------//136
- } ------------------------------------------------------//137
- Node merge(Node l, Node r) { ---------------------------//138
--- push_delta(l);    push_delta(r); ---------------------//139
--- if (!l || !r)   return l ? l : r; --------------------//13a
--- if (l->size <= r->size) { ----------------------------//13b
----- l->r = merge(l->r, r); -----------------------------//13c
----- update(l); -----------------------------------------//13d
----- return l; ------------------------------------------//13e
--- } else { ---------------------------------------------//13f
----- r->l = merge(l, r->l); -----------------------------//140
----- update(r); -----------------------------------------//141
----- return r; ------------------------------------------//142
--- } ----------------------------------------------------//143
- } ------------------------------------------------------//144
- void split(Node v, int key, Node &l, Node &r) { --------//145
--- push_delta(v); ---------------------------------------//146
--- l = r = NULL; ----------------------------------------//147
--- if (!v)     return; ----------------------------------//148
--- if (key <= get_size(v->l)) { -------------------------//149
----- split(v->l, key, l, v->l); -------------------------//14a
----- r = v; ---------------------------------------------//14b
--- } else { ---------------------------------------------//14c
----- split(v->r, key - get_size(v->l) - 1, v->r, r); ----//14d
----- l = v; ---------------------------------------------//14e
--- } ----------------------------------------------------//14f
--- update(v); -------------------------------------------//150
- } ------------------------------------------------------//151
- Node root; ---------------------------------------------//152
public: --------------------------------------------------//153
- cartree() : root(NULL) {} ------------------------------//154
- ~cartree() { delete root; } ----------------------------//155
- int get(Node v, int key) { -----------------------------//156
--- push_delta(v); ---------------------------------------//157
--- if (key < get_size(v->l)) ----------------------------//158
----- return get(v->l, key); -----------------------------//159
--- else if (key > get_size(v->l)) -----------------------//15a
----- return get(v->r, key - get_size(v->l) - 1); --------//15b
--- return v->node_val; ----------------------------------//15c
- } ------------------------------------------------------//15d
- int get(int key) { return get(root, key); } ------------//15e
- void insert(Node item, int key) { ----------------------//15f
--- Node l, r; -------------------------------------------//160
--- split(root, key, l, r); ------------------------------//161
--- root = merge(merge(l, item), r); ---------------------//162
- } ------------------------------------------------------//163
- void insert(int key, int val) { ------------------------//164
--- insert(new _Node(val), key); -------------------------//165
- } ------------------------------------------------------//166
- void erase(int key) { ----------------------------------//167
--- Node l, m, r; ----------------------------------------//168
--- split(root, key + 1, m, r); --------------------------//169
--- split(m, key, l, m); ---------------------------------//16a
--- delete m; --------------------------------------------//16b
--- root = merge(l, r); ----------------------------------//16c
- } ------------------------------------------------------//16d
- int query(int a, int b) { ------------------------------//16e
--- Node l1, r1; -----------------------------------------//16f
--- split(root, b+1, l1, r1); ----------------------------//170
--- Node l2, r2; -----------------------------------------//171
--- split(l1, a, l2, r2); --------------------------------//172
--- int res = get_subtree_val(r2); -----------------------//173
--- l1 = merge(l2, r2); ----------------------------------//174
--- root = merge(l1, r1); --------------------------------//175
--- return res; -------------------------------------------//176
- } ------------------------------------------------------//177
- int update(int a, int b, int delta) { ------------------//178
--- Node l1, r1; -----------------------------------------//179
--- split(root, b+1, l1, r1); ----------------------------//17a
--- Node l2, r2; -----------------------------------------//17b
--- split(l1, a, l2, r2); --------------------------------//17c
--- apply_delta(r2, delta); ------------------------------//17d
--- l1 = merge(l2, r2); ----------------------------------//17e
--- root = merge(l1, r1); --------------------------------//17f
- } ------------------------------------------------------//180
- int size() { return get_size(root); }  }; --------------//181
```

2.5.3. *Persistent Treap.*

2.6. **Ordered Statistics Tree.**

```
#include <ext/pb_ds/assoc_container.hpp> -----------------//049
#include <ext/pb_ds/tree_policy.hpp> ---------------------//04a
using namespace __gnu_pbds; ------------------------------//04b
template <typename T> ------------------------------------//04c
using indexed_set = std::tree<T, null_type, less<T>, -----//04d
splay_tree_tag, tree_order_statistics_node_update>; ------//04e
// indexed_set<int> t; t.insert(...); --------------------//04f
```

```
// t.find_by_order(index); // 0-based --------------------//050
// t.order_of_key(key); ---------------------------------//051
```

2.7. **Union Find.**

```
struct union_find { --------------------------------------//182
- vi p; union_find(int n) : p(n, -1) { } ----------------//183
- int find(int x) { return p[x] < 0 ? x : p[x] = find(p[x]); }
- bool unite(int x, int y) { ----------------------------//185
--- int xp = find(x), yp = find(y); ---------------------//186
--- if (xp == yp) return false; -------------------------//187
--- if (p[xp] > p[yp]) swap(xp,yp); ---------------------//188
--- p[xp] += p[yp], p[yp] = xp; -------------------------//189
--- return true; ----------------------------------------//18a
- } -----------------------------------------------------//18b
- int size(int x) { return -p[find(x)]; } ---------------//18c
}; -------------------------------------------------------//18d
```

3. Graphs

Using adjacency list:

```
struct graph { -------------------------------------------//1c4
- int n; -------------------------------------------------//1c5
- vii *adj; ----------------------------------------------//1c6
- int *dist; ---------------------------------------------//1c7
- graph(int n) { -----------------------------------------//1c8
--- this->n = n; -----------------------------------------//1c9
--- adj = new vii[n]; ------------------------------------//1ca
--- dist = new int[n]; -----------------------------------//1cb
- } ------------------------------------------------------//1cc
- void add_edge(int u, int v, int w) { -------------------//1cd
--- adj[u].push_back({v, w}); ----------------------------//1ce
--- /*adj[v].push_back({u, w});*/ ------------------------//1cf
- } ------------------------------------------------------//1d0
}; -------------------------------------------------------//1d1
```

Using adjacency matrix:

```
struct graph { -------------------------------------------//1d2
- int n; -------------------------------------------------//1d3
- int **mat; ---------------------------------------------//1d4
- graph(int n) { -----------------------------------------//1d5
--- this->n = n; -----------------------------------------//1d6
--- mat = new int*[n]; -----------------------------------//1d7
--- for (int i = 0; i < n; ++i) { ------------------------//1d8
----- mat[i] = new int[n]; -------------------------------//1d9
----- for (int j = 0; j < n; ++j) ------------------------//1da
------- mat[i][j] = INF; ---------------------------------//1db
----- mat[i][i] = 0; -------------------------------------//1dc
--- } ----------------------------------------------------//1dd
- } ------------------------------------------------------//1de
- void add_edge(int u, int v, int w) { -------------------//1df
--- mat[u][v] = std::min(mat[u][v], w); ------------------//1e0
- /*mat[v][u] = std::min(mat[v][u], w);*/ ---------------//1e1
- } ------------------------------------------------------//1e2
}; -------------------------------------------------------//1e3
```

Using edge list:

```
struct edge { --------------------------------------------//1e4
- int u, v, w; -------------------------------------------//1e5
- edge(int u, int v, int w) : u(u), v(v), w(w) {} -------//1e6
- const bool operator <(const edge &other) const { -------//1e7
--- return w < other.w; ----------------------------------//1e8
- } ------------------------------------------------------//1e9
}; -------------------------------------------------------//1ea
struct graph { -------------------------------------------//1eb
- int n; -------------------------------------------------//1ec
- std::vector<edge> edges; -------------------------------//1ed
- graph(int n) : n(n) {} ---------------------------------//1ee
- void add_edge(int u, int v, int w) { -------------------//1ef
--- edges.push_back(edge(u, v, w)); ----------------------//1f0
- } ------------------------------------------------------//1f1
}; -------------------------------------------------------//1f2
```

3.1. **Single-Source Shortest Paths.**

3.1.1. *Dijkstra.*

```
#include "graph_template_adjlist.cpp" --------------------//1a2
// insert inside graph; needs n, dist[], and adj[] -------//1a3
void dijkstra(int s) { -----------------------------------//1a4
- for (int u = 0; u < n; ++u) ----------------------------//1a5
--- dist[u] = INF; ---------------------------------------//1a6
- dist[s] = 0; -------------------------------------------//1a7
- std::priority_queue<ii, vii, std::greater<ii> > pq; ----//1a8
- pq.push({0, s}); ---------------------------------------//1a9
- while (!pq.empty()) { ----------------------------------//1aa
--- int u = pq.top().second; -----------------------------//1ab
--- int d = pq.top().first; ------------------------------//1ac
--- pq.pop(); --------------------------------------------//1ad
--- if (dist[u] < d) -------------------------------------//1ae
----- continue; ------------------------------------------//1af
--- dist[u] = d; -----------------------------------------//1b0
--- for (auto &e : adj[u]) { -----------------------------//1b1
----- int v = e.first; -----------------------------------//1b2
----- int w = e.second; ----------------------------------//1b3
----- if (dist[v] > dist[u] + w) { -----------------------//1b4
------- dist[v] = dist[u] + w; ---------------------------//1b5
------- pq.push({dist[v], v}); ---------------------------//1b6
----- } --------------------------------------------------//1b7
--- } ----------------------------------------------------//1b8
- } ------------------------------------------------------//1b9
} --------------------------------------------------------//1ba
```

3.1.2. *Bellman-Ford.*

```
#include "graph_template_adjlist.cpp" --------------------//18e
// insert inside graph; needs n, dist[], and adj[] -------//18f
void bellman_ford(int s) { -------------------------------//190
- for (int u = 0; u < n; ++u) ----------------------------//191
--- dist[u] = INF; ---------------------------------------//192
- dist[s] = 0; -------------------------------------------//193
- for (int i = 0; i < n-1; ++i) --------------------------//194
--- for (int u = 0; u < n; ++u) --------------------------//195
----- for (auto &e : adj[u]) -----------------------------//196
------- if (dist[u] + e.second < dist[e.first]) ----------//197
--------- dist[e.first] = dist[u] + e.second; ------------//198
} --------------------------------------------------------//199
// you can call this after running bellman_ford() --------//19a
bool has_neg_cycle() { -----------------------------------//19b
- for (int u = 0; u < n; ++u) ----------------------------//19c
--- for (auto &e : adj[u]) -------------------------------//19d
----- if (dist[e.first] > dist[u] + e.second) ------------//19e
------- return true; -------------------------------------//19f
- return false; ------------------------------------------//1a0
} --------------------------------------------------------//1a1
```

3.1.3. *SPFA.*

3.2. **All-Pairs Shortest Paths.**

3.2.1. *Floyd-Washall.*

```
#include "graph_template_adjmat.cpp" ---------------------//1bb
// insert inside graph; needs n and mat[][] --------------//1bc
void floyd_warshall() { ----------------------------------//1bd
- for (int k = 0; k < n; ++k) ----------------------------//1be
--- for (int i = 0; i < n; ++i) --------------------------//1bf
----- for (int j = 0; j < n; ++j) ------------------------//1c0
------- if (mat[i][k] + mat[k][j] < mat[i][j]) -----------//1c1
--------- mat[i][j] = mat[i][k] + mat[k][j]; -------------//1c2
} --------------------------------------------------------//1c3
```

3.3. **Strongly Connected Components.**

3.3.1. *Kosaraju.*

```
struct kosaraju_graph { ----------------------------------//23c
- int n; -------------------------------------------------//23d
- int *vis; ----------------------------------------------//23e
- vi **adj; ----------------------------------------------//23f
- std::vector<vi> sccs; ----------------------------------//240
- kosaraju_graph(int n) { --------------------------------//241
--- this->n = n; -----------------------------------------//242
--- vis = new int[n]; ------------------------------------//243
--- adj = new vi*[2]; ------------------------------------//244
--- for (int dir = 0; dir < 2; ++dir) --------------------//245
----- adj[dir] = new vi[n]; ------------------------------//246
- } ------------------------------------------------------//247
- void add_edge(int u, int v) { --------------------------//248
--- adj[0][u].push_back(v); ------------------------------//249
--- adj[1][v].push_back(u); ------------------------------//24a
- } ------------------------------------------------------//24b
- void dfs(int u, int p, int dir, vi &topo) { ------------//24c
--- vis[u] = 1; ------------------------------------------//24d
--- for (int v : adj[dir][u]) ----------------------------//24e
----- if (!vis[v] && v != p) -----------------------------//24f
------- dfs(v, u, dir, topo); ----------------------------//250
--- topo.push_back(u); -----------------------------------//251
- } ------------------------------------------------------//252
- void kosaraju() { --------------------------------------//253
--- vi topo; ---------------------------------------------//254
--- for (int u = 0; u < n; ++u)  vis[u] = 0; -------------//255
--- for (int u = 0; u < n; ++u) --------------------------//256
----- if (!vis[u]) ---------------------------------------//257
------- dfs(u, -1, 0, topo); -----------------------------//258
--- for (int u = 0; u < n; ++u)  vis[u] = 0; -------------//259
--- for (int i = n-1; i >= 0; --i) { ---------------------//25a
----- if (!vis[topo[i]]) { -------------------------------//25b
```

```
------- sccs.push_back({}); -----------------------------//25c
------- dfs(topo[i], -1, 1, sccs.back()); ---------------//25d
----- } -------------------------------------------------//25e
--- } ---------------------------------------------------//25f
- } -----------------------------------------------------//260
}; ------------------------------------------------------//261
```

3.4. Cut Points and Bridges.

3.5. Biconnected Components.

3.5.1. *Bridge Tree.*

3.5.2. *Block-Cut Tree.*

3.6. Minimum Spanning Tree.

3.6.1. *Kruskal.*

3.6.2. *Prim.*

3.7. Topological Sorting.

3.8. Euler Path.

3.9. Bipartite Matching.

3.9.1. *Alternating Paths Algorithm.*

3.9.2. *Hopcroft-Karp Algorithm.*

3.10. Maximum Flow.

3.10.1. *Edmonds-Karp.*

```
struct flow_network { -----------------------------------//2e4
- int n, s, t, *par, **c, **f; --------------------------//2e5
- vi *adj; ----------------------------------------------//2e6
- flow_network(int n, int s, int t) : n(n), s(s), t(t) { -//2e7
--- adj = new std::vector<int>[n]; ----------------------//2e8
--- par = new int[n]; -----------------------------------//2e9
--- c = new int*[n]; ------------------------------------//2ea
--- f = new int*[n]; ------------------------------------//2eb
--- for (int i = 0; i < n; ++i) { -----------------------//2ec
----- c[i] = new int[n]; --------------------------------//2ed
----- f[i] = new int[n]; --------------------------------//2ee
----- for (int j = 0; j < n; ++j) -----------------------//2ef
------- c[i][j] = f[i][j] = 0; --------------------------//2f0
--- } ---------------------------------------------------//2f1
- } -----------------------------------------------------//2f2
- void add_edge(int u, int v, int w) { ------------------//2f3
--- adj[u].push_back(v); --------------------------------//2f4
--- adj[v].push_back(u); --------------------------------//2f5
--- c[u][v] += w; ---------------------------------------//2f6
- } -----------------------------------------------------//2f7
- int res(int i, int j) { return c[i][j] - f[i][j]; } ----//2f8
- bool bfs() { ------------------------------------------//2f9
--- std::queue<int> q; ----------------------------------//2fa
--- q.push(this->s); ------------------------------------//2fb
--- while (!q.empty()) { --------------------------------//2fc
----- int u = q.front();  q.pop(); ----------------------//2fd
----- for (int v : adj[u]) { ----------------------------//2fe
------- if (res(u, v) > 0 and par[v] == -1) { -----------//2ff
--------- par[v] = u; -----------------------------------//300
--------- if (v == this->t) -----------------------------//301
----------- return true; --------------------------------//302
--------- q.push(v); ------------------------------------//303
------- } -----------------------------------------------//304
----- } -------------------------------------------------//305
--- } ---------------------------------------------------//306
--- return false; ---------------------------------------//307
- } -----------------------------------------------------//308
- bool aug_path() { -------------------------------------//309
--- for (int u = 0; u < n; ++u) -------------------------//30a
----- par[u] = -1; --------------------------------------//30b
--- par[s] = s; -----------------------------------------//30c
--- return bfs(); ---------------------------------------//30d
- } -----------------------------------------------------//30e
- int calc_max_flow() { ---------------------------------//30f
--- int ans = 0; ----------------------------------------//310
--- while (aug_path()) { --------------------------------//311
----- int flow = INF; -----------------------------------//312
----- for (int u = t; u != s; u = par[u]) ---------------//313
------- flow = std::min(flow, res(par[u], u)); ----------//314
----- for (int u = t; u != s; u = par[u]) ---------------//315
------- f[par[u]][u] += flow, f[u][par[u]] -= flow; ------//316
----- ans += flow; --------------------------------------//317
--- } ---------------------------------------------------//318
--- return ans; -----------------------------------------//319
- } -----------------------------------------------------//31a
}; ------------------------------------------------------//31b
```

3.10.2. *Dinic.*

```
struct flow_network { -----------------------------------//296
- int n, s, t, *adj_ptr, *dist, *par, **c, **f; ---------//297
- vi *adj; ----------------------------------------------//298
- flow_network(int n, int s, int t) : n(n), s(s), t(t) { -//299
--- adj = new std::vector<int>[n]; ----------------------//29a
--- adj_ptr = new int[n]; -------------------------------//29b
--- dist = new int[n]; ----------------------------------//29c
--- par = new int[n]; -----------------------------------//29d
--- c = new int*[n]; ------------------------------------//29e
--- f = new int*[n]; ------------------------------------//29f
--- for (int i = 0; i < n; ++i) { -----------------------//2a0
----- c[i] = new int[n]; --------------------------------//2a1
----- f[i] = new int[n]; --------------------------------//2a2
----- for (int j = 0; j < n; ++j) -----------------------//2a3
------- c[i][j] = f[i][j] = 0; --------------------------//2a4
--- } ---------------------------------------------------//2a5
- } -----------------------------------------------------//2a6
- void add_edge(int u, int v, int w) { ------------------//2a7
--- adj[u].push_back(v); --------------------------------//2a8
--- adj[v].push_back(u); --------------------------------//2a9
--- c[u][v] += w; ---------------------------------------//2aa
- } -----------------------------------------------------//2ab
- int res(int i, int j) { return c[i][j] - f[i][j]; } ----//2ac
- void reset(int *ar, int val) { ------------------------//2ad
--- for (int i = 0; i < n; ++i) -------------------------//2ae
----- ar[i] = val; --------------------------------------//2af
- } -----------------------------------------------------//2b0
- bool make_level_graph() { -----------------------------//2b1
--- reset(dist, -1); ------------------------------------//2b2
--- std::queue<int> q; ----------------------------------//2b3
--- q.push(s); ------------------------------------------//2b4
--- dist[s] = 0; ----------------------------------------//2b5
--- while (!q.empty()) { --------------------------------//2b6
----- int u = q.front();  q.pop(); ----------------------//2b7
----- for (int v : adj[u]) { ----------------------------//2b8
------- if (res(u, v) > 0 and dist[v] == -1) { ----------//2b9
--------- dist[v] = dist[u] + 1; ------------------------//2ba
--------- q.push(v); ------------------------------------//2bb
------- } -----------------------------------------------//2bc
----- } -------------------------------------------------//2bd
--- } ---------------------------------------------------//2be
--- return dist[t] != -1; -------------------------------//2bf
- } -----------------------------------------------------//2c0
- bool next(int u, int v) { -----------------------------//2c1
--- return dist[v] == dist[u] + 1; ----------------------//2c2
- } -----------------------------------------------------//2c3
- bool dfs(int u) { -------------------------------------//2c4
--- if (u == t)   return true; --------------------------//2c5
--- for (int &i = adj_ptr[u]; i < adj[u].size(); ++i) { --//2c6
----- int v = adj[u][i]; --------------------------------//2c7
----- if (next(u, v) and res(u, v) > 0 and dfs(v)) { -----//2c8
------- par[v] = u; -------------------------------------//2c9
------- return true; ------------------------------------//2ca
----- } -------------------------------------------------//2cb
--- } ---------------------------------------------------//2cc
--- dist[u] = -1; ---------------------------------------//2cd
--- return false; ---------------------------------------//2ce
- } -----------------------------------------------------//2cf
- bool aug_path() { -------------------------------------//2d0
--- reset(par, -1); -------------------------------------//2d1
--- par[s] = s; -----------------------------------------//2d2
--- return dfs(s);  } -----------------------------------//2d3
- int calc_max_flow() { ---------------------------------//2d4
--- int ans = 0; ----------------------------------------//2d5
--- while (make_level_graph()) { ------------------------//2d6
----- reset(adj_ptr, 0); --------------------------------//2d7
----- while (aug_path()) { ------------------------------//2d8
------- int flow = INF; ---------------------------------//2d9
------- for (int u = t; u != s; u = par[u]) -------------//2da
--------- flow = std::min(flow, res(par[u], u)); --------//2db
------- for (int u = t; u != s; u = par[u]) -------------//2dc
--------- f[par[u]][u] += flow, f[u][par[u]] -= flow; ----//2dd
------- ans += flow; ------------------------------------//2de
----- } -------------------------------------------------//2df
--- } ---------------------------------------------------//2e0
--- return ans; -----------------------------------------//2e1
- } -----------------------------------------------------//2e2
}; ------------------------------------------------------//2e3
```

3.11. All-pairs Maximum Flow.

3.11.1. *Gomory-Hu.*

3.12. Heavy Light Decomposition.

```
#include "segment_tree.cpp" ------------------------------//1f3
struct heavy_light_tree { --------------------------------//1f4
- int n; -------------------------------------------------//1f5
- std::vector<int> *adj; ---------------------------------//1f6
- segtree *segment_tree; ---------------------------------//1f7
- int *par, *heavy, *dep, *path_root, *pos; --------------//1f8
- heavy_light_tree(int n) { ------------------------------//1f9
--- this->n = n; -----------------------------------------//1fa
--- this->adj = new std::vector<int>[n]; -----------------//1fb
--- segment_tree = new segtree(0, n-1); ------------------//1fc
--- par = new int[n]; ------------------------------------//1fd
--- heavy = new int[n]; ----------------------------------//1fe
--- dep = new int[n]; ------------------------------------//1ff
--- path_root = new int[n]; ------------------------------//200
--- pos = new int[n]; ------------------------------------//201
- } ------------------------------------------------------//202
- void add_edge(int u, int v) { --------------------------//203
--- adj[u].push_back(v); ---------------------------------//204
--- adj[v].push_back(u); ---------------------------------//205
- } ------------------------------------------------------//206
- void build(int root) { ---------------------------------//207
--- for (int u = 0; u < n; ++u) { ------------------------//208
----- heavy[u] = -1; -------------------------------------//209
--- par[root] = root; ------------------------------------//20a
--- dep[root] = 0; ---------------------------------------//20b
--- dfs(root); -------------------------------------------//20c
--- for (int u = 0, p = 0; u < n; ++u) { -----------------//20d
----- if (par[u] == -1 or heavy[par[u]] != u) { ----------//20e
------- for (int v = u; v != -1; v = heavy[v]) { ---------//20f
--------- path_root[v] = u; ------------------------------//210
--------- pos[v] = p++; ----------------------------------//211
------- } ------------------------------------------------//212
----- } --------------------------------------------------//213
--- } ----------------------------------------------------//214
- } ------------------------------------------------------//215
- int dfs(int u) { ---------------------------------------//216
--- int sz = 1; ------------------------------------------//217
--- int max_subtree_sz = 0; ------------------------------//218
--- for (int v : adj[u]) { -------------------------------//219
----- if (v != par[u]) { ---------------------------------//21a
------- par[v] = u; --------------------------------------//21b
------- dep[v] = dep[u] + 1; -----------------------------//21c
------- int subtree_sz = dfs(v); -------------------------//21d
------- if (max_subtree_sz < subtree_sz) { ---------------//21e
--------- max_subtree_sz = subtree_sz; -------------------//21f
--------- heavy[u] = v; ----------------------------------//220
------- } ------------------------------------------------//221
------- sz += subtree_sz; --------------------------------//222
----- } --------------------------------------------------//223
--- } ----------------------------------------------------//224
--- return sz; -------------------------------------------//225
- } ------------------------------------------------------//226
- int query(int u, int v) { ------------------------------//227
--- int res = 0; -----------------------------------------//228
--- while (path_root[u] != path_root[v]) { ---------------//229
----- if (dep[path_root[u]] > dep[path_root[v]]) ---------//22a
------- std::swap(u, v); ---------------------------------//22b
----- res += segment_tree->sum(pos[path_root[v]], pos[v]);//22c
----- v = par[path_root[v]]; -----------------------------//22d
--- } ----------------------------------------------------//22e
--- res += segment_tree->sum(pos[u], pos[v]); ------------//22f
--- return res; ------------------------------------------//230
- } ------------------------------------------------------//231
- void update(int u, int v, int c) { ---------------------//232
--- for (; path_root[u] != path_root[v]; -----------------//233
--------- v = par[path_root[v]]) { -----------------------//234
----- if (dep[path_root[u]] > dep[path_root[v]]) ---------//235
------- std::swap(u, v); ---------------------------------//236
----- segment_tree->increase(pos[path_root[v]], pos[v], c);
--- } ----------------------------------------------------//238
--- segment_tree->increase(pos[u], pos[v], c); -----------//239
- } ------------------------------------------------------//23a
}; -------------------------------------------------------//23b
```

3.13. Centroid Decomposition.

3.14. Least Common Ancestor.

3.14.1. *Binary Lifting.*

```
struct graph { -------------------------------------------//262
- int n; -------------------------------------------------//263
- int logn; ----------------------------------------------//264
- std::vector<int> *adj; ---------------------------------//265
- int *dep; ----------------------------------------------//266
- int **par; ---------------------------------------------//267
- graph(int n, int logn=20) { ----------------------------//268
--- this->n = n; -----------------------------------------//269
--- this->logn = logn; -----------------------------------//26a
--- adj = new std::vector<int>[n]; -----------------------//26b
--- dep = new int[n]; ------------------------------------//26c
--- par = new int*[n]; -----------------------------------//26d
--- for (int i = 0; i < n; ++i) --------------------------//26e
----- par[i] = new int[logn]; ----------------------------//26f
- } ------------------------------------------------------//270
- void dfs(int u, int p, int d) { ------------------------//271
--- dep[u] = d; ------------------------------------------//272
--- par[u][0] = p; ---------------------------------------//273
--- for (int v : adj[u]) ---------------------------------//274
----- if (v != p) ----------------------------------------//275
------- dfs(v, u, d+1); ----------------------------------//276
- } ------------------------------------------------------//277
- int ascend(int u, int k) { -----------------------------//278
--- for (int i = 0; i < logn; ++i) -----------------------//279
----- if (k & (1 << i)) ----------------------------------//27a
------- u = par[u][i]; -----------------------------------//27b
--- return u; --------------------------------------------//27c
- } ------------------------------------------------------//27d
- int lca(int u, int v) { --------------------------------//27e
--- if (dep[u] > dep[v])  u = ascend(u, dep[u] - dep[v]); //27f
--- if (dep[v] > dep[u])  v = ascend(v, dep[v] - dep[u]); //280
--- if (u == v)           return u; ----------------------//281
--- for (int k = logn-1; k >= 0; --k) { ------------------//282
----- if (par[u][k] != par[v][k]) { ----------------------//283
------- u = par[u][k]; -----------------------------------//284
------- v = par[v][k]; -----------------------------------//285
----- } --------------------------------------------------//286
--- } ----------------------------------------------------//287
--- return par[u][0]; ------------------------------------//288
- } ------------------------------------------------------//289
- bool is_anc(int u, int v) { ----------------------------//28a
--- if (dep[u] < dep[v]) ---------------------------------//28b
----- std::swap(u, v); -----------------------------------//28c
--- return ascend(u, dep[u] - dep[v]) == v; --------------//28d
- } ------------------------------------------------------//28e
- void prep_lca(int root=0) { ----------------------------//28f
--- dfs(root, root, 0); ----------------------------------//290
--- for (int k = 1; k < logn; ++k) -----------------------//291
----- for (int u = 0; u < n; ++u) ------------------------//292
------- par[u][k] = par[par[u][k-1]][k-1]; ---------------//293
- } ------------------------------------------------------//294
}; -------------------------------------------------------//295
```

3.14.2. *Tarjan's Offline Algorithm.*

4. Strings

4.1. Z-algorithm.

4.2. Trie.

4.3. Hashing.

4.3.1. *Polynomial Hashing.*

```
int MAXN = 1e5+1, MOD = 1e9+7; ---------------------------//35b
struct hasher { ------------------------------------------//35c
- int n; -------------------------------------------------//35d
- std::vector<ll> *p_pow; --------------------------------//35e
- std::vector<ll> *h_ans; --------------------------------//35f
- hash(vi &s, vi primes) { -------------------------------//360
--- n = primes.size(); -----------------------------------//361
--- p_pow = new std::vector<ll>[n]; ----------------------//362
--- h_ans = new std::vector<ll>[n]; ----------------------//363
--- for (int i = 0; i < n; ++i) { ------------------------//364
----- p_pow[i] = std::vector<ll>(MAXN); ------------------//365
----- p_pow[i][0] = 1; -----------------------------------//366
----- for (int j = 0; j+1 < MAXN; ++j) -------------------//367
------- p_pow[i][j+1] = (p_pow[i][j] * primes[i]) % MOD; -//368
----- h_ans[i] = std::vector<ll>(MAXN); ------------------//369
----- h_ans[i][0] = 0; -----------------------------------//36a
----- for (int j = 0; j < s.size(); ++j) -----------------//36b
------- h_ans[i][j+1] = (h_ans[i][j] + -------------------//36c
------------------------ s[j] * p_pow[i][j]) % MOD; ------//36d
--- } ----------------------------------------------------//36e
- } ------------------------------------------------------//36f
}; -------------------------------------------------------//370
```

5. Dynamic Programming

5.1. Longest Common Subsequence.

5.2. Longest Increasing Subsequence.

5.3. Traveling Salesman.

6. Mathematics

6.1. Special Data Types.

6.1.1. *Fraction.*

6.1.2. *BigInteger.*

6.1.3. *Matrix.*

6.1.4. *Dates.*

6.2. Binomial Coefficients.

6.3. Euclidean Algorithm.

6.4. Primality Test.

6.4.1. *Optimized Brute Force.*

6.4.2. *Miller-Rabin.*

6.4.3. *Pollard's Rho Algorithm.*

6.5. Sieve.

6.5.1. *Sieve of Eratosthenes.*

6.5.2. *Divisor Sieve (Modified Sieve of Eratosthenes).*

6.5.3. *Phi Sieve.*

6.6. Phi Function.

6.7. Modular Exponentiation.

```
ll fast_exp(ll base, ll exp, ll mod) {
- ll res = 1;
- while (exp) {
--- if (exp & 1)  res = (res * base) % mod;
--- base = (base * base) % mod;
--- exp >>= 1;
- }
- return res % mod;
}
```

6.8. Modular Multiplicative Inverse.

```
ll mod_inverse(ll x, ll mod) {
- return fast_exp(x, mod-2, mod);
}
```

6.9. Chinese Remainder Theorem.

```
typedef std::pair<ll, ll> pll;
ll crt(std::vector<pll> &data) {
- ll M = 1, res = 0;
- for (int i = 0; i < data.size(); ++i)
--- M *= data[i].second;
- for (int i = 0; i < data.size(); ++i) {
--- ll Mi = M/data[i].second;
--- res = (res + data[i].first * Mi *
--------------- mod_inverse(Mi, data[i].second)) % M;
- }
- return res;
}
pll crt_generalized(std::vector<pll> &data) {
- std::map<ll, pll> decomp;
- for (int i = 0; i < data.size(); ++i) {
--- ll m = data[i].second;
--- for (ll f = 2; f*f <= m; f = f == 2 ? 3 : f + 2) {
----- ll cur = 1;
----- while (m % f == 0) {
------- m /= f;
------- cur *= f;
----- }
----- if (cur > 1 and cur > decomp[f].second)
------- decomp[f] = {data[i].first % cur, cur};
--- }
--- if (m > 1 and m > decomp[m].second)
----- decomp[m] = {data[i].first % m, m};
- }
- std::vector<pll> new_data;
- ll M = 1;
- for (auto &[f, cm] : decomp) {
----- new_data.push_back(cm);
----- M *= cm.second;
- }
- ll x = crt(new_data);
- for (int i = 0; i < data.size(); ++i)
--- if (x % M != data[i].first % data[i].second)
----- return {0, 0};
- return {x, M};
}
ll crt_test(std::vector<pll> &data) {
- if (data.size() <= 1)
--- return std::__gcd(data[0].first, data[0].second) == 1;
- ll gC = std::__gcd(data[0].first, data[1].first);
- ll gM = std::__gcd(data[0].second, data[1].second);
- for (int i = 2; i < data.size(); ++i) {
--- gC = std::__gcd(gC, data[i].first);
--- gM = std::__gcd(gM, data[i].second);
- }
- return (gC % gM) == 0;
}
```

6.10. Numeric Integration (Simpson's Rule).

6.11. Fast Fourier Transform.

6.12. Josephus Problem.

6.13. Number of Integer Points Below a Line.

7. Geometry

7.1. Primitives.

7.2. Lines.

7.3. Circles.

7.4. Polygons.

7.5. Convex Hull (Graham's Scan).

7.6. Closest Pair of Points.

7.7. Rectilinear Minimum Spanning Tree.

8. Other Algorithms

8.1. Coordinate Compression.

8.2. 2SAT.

8.3. Nth Permutation.

8.4. Floyd's Cycle-Finding.

8.5. Simulated Annealing.

8.6. Hexagonal Grid Algorithms.

9. Useful Information (CLEAN THIS UP!!)

10. Misc

10.1. **Debugging Tips.**

- Stack overflow? Recursive DFS on tree that is actually a long path?
- Floating-point numbers
  - Getting `NaN`? Make sure `acos` etc. are not getting values out of their range (perhaps `1+eps`).
  - Rounding negative numbers?
  - Outputting in scientific notation?
- Wrong Answer?
  - Read the problem statement again!
  - Are multiple test cases being handled correctly? Try repeating the same test case many times.
  - Integer overflow?
  - Think very carefully about boundaries of all input parameters
  - Try out possible edge cases:
    * $n = 0, n = -1, n = 1, n = 2^{31} - 1$ or $n = -2^{31}$
    * List is empty, or contains a single element
    * n is even, n is odd
    * Graph is empty, or contains a single vertex
    * Graph is a multigraph (loops or multiple edges)
    * Polygon is concave or non-simple
  - Is initial condition wrong for small cases?
  - Are you sure the algorithm is correct?
  - Explain your solution to someone.
  - Are you using any functions that you don't completely understand? Maybe STL functions?
  - Maybe you (or someone else) should rewrite the solution?
  - Can the input line be empty?
- Run-Time Error?
  - Is it actually Memory Limit Exceeded?

10.2. **Solution Ideas.**

- Dynamic Programming
  - Parsing CFGs: CYK Algorithm
  - Drop a parameter, recover from others
  - Swap answer and a parameter
  - When grouping: try splitting in two
  - 2^k trick
  - When optimizing
    * Convex hull optimization
      · $\mathrm{dp}[i] = \min_{j<i}\{\mathrm{dp}[j] + b[j] \times a[i]\}$
      · $b[j] \geq b[j+1]$
      · optionally $a[i] \leq a[i+1]$
      · $O(n^2)$ to $O(n)$
    * Divide and conquer optimization
      · $\mathrm{dp}[i][j] = \min_{k<j}\{\mathrm{dp}[i-1][k] + C[k][j]\}$
      · $A[i][j] \leq A[i][j+1]$
      · $O(kn^2)$ to $O(kn \log n)$
      · sufficient: $C[a][c] + C[b][d] \leq C[a][d] + C[b][c]$, $a \leq b \leq c \leq d$ (QI)
    * Knuth optimization
      · $\mathrm{dp}[i][j] = \min_{i<k<j}\{\mathrm{dp}[i][k] + \mathrm{dp}[k][j] + C[i][j]\}$
      · $A[i][j-1] \leq A[i][j] \leq A[i+1][j]$
      · $O(n^3)$ to $O(n^2)$
      · sufficient: QI and $C[b][c] \leq C[a][d]$, $a \leq b \leq c \leq d$
- Greedy
- Randomized
- Optimizations
  - Use bitset (/64)
  - Switch order of loops (cache locality)
- Process queries offline
  - Mo's algorithm
- Square-root decomposition
- Precomputation
- Efficient simulation
  - Mo's algorithm
  - Sqrt decomposition
  - Store 2^k jump pointers
- Data structure techniques
  - Sqrt buckets
  - Store 2^k jump pointers
  - 2^k merging trick
- Counting
  - Inclusion-exclusion principle
  - Generating functions
- Graphs
  - Can we model the problem as a graph?
  - Can we use any properties of the graph?
  - Strongly connected components
  - Cycles (or odd cycles)
  - Bipartite (no odd cycles)
    * Bipartite matching
    * Hall's marriage theorem
    * Stable Marriage
  - Cut vertex/bridge
  - Biconnected components
  - Degrees of vertices (odd/even)
  - Trees
    * Heavy-light decomposition
    * Centroid decomposition
    * Least common ancestor
    * Centers of the tree
  - Eulerian path/circuit
  - Chinese postman problem
  - Topological sort
  - (Min-Cost) Max Flow
  - Min Cut
    * Maximum Density Subgraph
  - Huffman Coding
  - Min-Cost Arborescence
  - Steiner Tree
  - Kirchoff's matrix tree theorem
  - Prüfer sequences
  - Lovász Toggle
  - Look at the DFS tree (which has no cross-edges)
  - Is the graph a DFA or NFA?
    * Is it the Synchronizing word problem?
- Mathematics
  - Is the function multiplicative?
  - Look for a pattern
  - Permutations
    * Consider the cycles of the permutation
  - Functions
    * Sum of piecewise-linear functions is a piecewise-linear function
    * Sum of convex (concave) functions is convex (concave)
  - Modular arithmetic
    * Chinese Remainder Theorem
    * Linear Congruence
  - Sieve
  - System of linear equations
  - Values too big to represent?
    * Compute using the logarithm
    * Divide everything by some large value
  - Linear programming
    * Is the dual problem easier to solve?
  - Can the problem be modeled as a different combinatorial problem? Does that simplify calculations?
- Logic
  - 2-SAT
  - XOR-SAT (Gauss elimination or Bipartite matching)
- Meet in the middle
- Only work with the smaller half ($\log(n)$)
- Strings
  - Trie (maybe over something weird, like bits)
  - Suffix array
  - Suffix automaton (+DP?)
  - Aho-Corasick
  - eerTree
  - Work with $S + S$
- Hashing
- Euler tour, tree to array
- Segment trees
  - Lazy propagation
  - Persistent
  - Implicit
  - Segment tree of X
- Geometry
  - Minkowski sum (of convex sets)
  - Rotating calipers
  - Sweep line (horizontally or vertically?)
  - Sweep angle
  - Convex hull
- Fix a parameter (possibly the answer).
- Are there few distinct values?
- Binary search
- Sliding Window (+ Monotonic Queue)
- Computing a Convolution? Fast Fourier Transform
- Computing a 2D Convolution? FFT on each row, and then on each column
- Exact Cover (+ Algorithm X)
- Cycle-Finding
- What is the smallest set of values that identify the solution? The cycle structure of the permutation? The powers of primes in the factorization?
- Look at the complement problem

- Minimize something instead of maximizing
- Immediately enforce necessary conditions. (All values greater than 0? Initialize them all to 1)
- Add large constant to negative numbers to make them positive
- Counting/Bucket sort

11. Formulas

- **Legendre symbol:** $\left(\frac{a}{b}\right) = a^{(b-1)/2} \pmod b$, b odd prime.
- **Heron's formula:** A triangle with side lengths a, b, c has area $\sqrt{s(s-a)(s-b)(s-c)}$ where $s = \frac{a+b+c}{2}$.
- **Pick's theorem:** A polygon on an integer grid strictly containing i lattice points and having b lattice points on the boundary has area $i + \frac{b}{2} - 1$. (Nothing similar in higher dimensions)
- **Euler's totient:** The number of integers less than n that are coprime to n are $n\prod_{p|n}\left(1-\frac{1}{p}\right)$ where each p is a distinct prime factor of n.
- **König's theorem:** In any bipartite graph $G = (L \cup R, E)$, the number of edges in a maximum matching is equal to the number of vertices in a minimum vertex cover. Let U be the set of unmatched vertices in L, and Z be the set of vertices that are either in U or are connected to U by an alternating path. Then $K = (L \setminus Z) \cup (R \cap Z)$ is the minimum vertex cover.
- A minumum Steiner tree for n vertices requires at most $n-2$ additional Steiner vertices.
- The number of vertices of a graph is equal to its minimum vertex cover number plus the size of a maximum independent set.
- **Lagrange polynomial** through points $(x_0, y_0), \ldots, (x_k, y_k)$ is $L(x) = \sum_{j=0}^{k} y_j \prod_{\substack{0 \le m \le k \\ m \neq j}} \frac{x - x_m}{x_j - x_m}$
- **Hook length formula:** If λ is a Young diagram and $h_\lambda(i,j)$ is the hook-length of cell (i,j), then then the number of Young tableux $d_\lambda = n!/\prod h_\lambda(i,j)$.
- **Möbius inversion formula:** If $f(n) = \sum_{d|n} g(d)$, then $g(n) = \sum_{d|n} \mu(d) f(n/d)$. If $f(n) = \sum_{m=1}^{n} g(\lfloor n/m \rfloor)$, then $g(n) = \sum_{m=1}^{n} \mu(m) f(\lfloor \frac{n}{m} \rfloor)$.
- #primitive pythagorean triples with hypotenuse $< n$ approx $n/(2\pi)$.
- **Frobenius Number:** largest number which can't be expressed as a linear combination of numbers $a_1, \ldots, a_n$ with non-negative coefficients. $g(a_1, a_2) = a_1 a_2 - a_1 - a_2$, $N(a_1, a_2) = (a_1 - 1)(a_2 - 1)/2$. $g(d \cdot a_1, d \cdot a_2, a_3) = d \cdot g(a_1, a_2, a_3) + a_3(d-1)$. An integer $x > (\max_i a_i)^2$ can be expressed in such a way iff. $x \mid \gcd(a_1, \ldots, a_n)$.

11.1. Physics.

- **Snell's law:** $\frac{\sin\theta_1}{v_1} = \frac{\sin\theta_2}{v_2}$

11.2. Markov Chains.

A Markov Chain can be represented as a weighted directed graph of states, where the weight of an edge represents the probability of transitioning over that edge in one timestep. Let $P^{(m)} = (p_{ij}^{(m)})$ be the probability matrix of transitioning from state i to state j in m timesteps, and note that $P^{(1)}$ is the adjacency matrix of the graph. **Chapman-Kolmogorov:** $p_{ij}^{(m+n)} = \sum_k p_{ik}^{(m)} p_{kj}^{(n)}$. It follows that $P^{(m+n)} = P^{(m)} P^{(n)}$ and $P^{(m)} = P^m$. If $p^{(0)}$ is the initial probability distribution (a vector), then $p^{(0)} P^{(m)}$ is the probability distribution after m timesteps.

The return times of a state i is $R_i = \{m \mid p_{ii}^{(m)} > 0\}$, and i is *aperiodic* if $\gcd(R_i) = 1$. A MC is aperiodic if any of its vertices is aperiodic. A MC is *irreducible* if the corresponding graph is strongly connected.

A distribution π is stationary if $\pi P = \pi$. If MC is irreducible then $\pi_i = 1/\mathbb{E}[T_i]$, where T_i is the expected time between two visits at i. π_j/π_i is the expected number of visits at j in between two consecutive visits at i. A MC is *ergodic* if $\lim_{m\to\infty} p^{(0)} P^m = \pi$. A MC is ergodic iff. it is irreducible and aperiodic.

A MC for a random walk in an undirected weighted graph (unweighted graph can be made weighted by adding 1-weights) has $p_{uv} = w_{uv}/\sum_x w_{ux}$. If the graph is connected, then $\pi_u = \sum_x w_{ux}/\sum_v \sum_x w_{vx}$. Such a random walk is aperiodic iff. the graph is not bipartite.

An *absorbing* MC is of the form $P = \begin{pmatrix} Q & R \\ 0 & I_r \end{pmatrix}$. Let $N = \sum_{m=0}^{\infty} Q^m = (I_t - Q)^{-1}$. Then, if starting in state i, the expected number of steps till absorption is the i-th entry in $N1$. If starting in state i, the probability of being absorbed in state j is the (i,j)-th entry of NR.

Many problems on MC can be formulated in terms of a system of recurrence relations, and then solved using Gaussian elimination.

11.3. Burnside's Lemma.

Let G be a finite group that acts on a set X. For each g in G let X^g denote the set of elements in X that are fixed by g. Then the number of orbits

$$|X/G| = \frac{1}{|G|} \sum_{g \in G} |X^g|$$

$$Z(S_n) = \frac{1}{n} \sum_{l=1}^{n} a_l Z(S_{n-l})$$

11.4. Bézout's identity.

If (x, y) is any solution to $ax + by = d$ (e.g. found by the Extended Euclidean Algorithm), then all solutions are given by

$$\left(x + k\frac{b}{\gcd(a,b)}, y - k\frac{a}{\gcd(a,b)}\right)$$

11.5. Misc.

11.5.1. *Determinants and PM.*

$$det(A) = \sum_{\sigma \in S_n} \operatorname{sgn}(\sigma) \prod_{i=1}^{n} a_{i,\sigma(i)}$$

$$perm(A) = \sum_{\sigma \in S_n} \prod_{i=1}^{n} a_{i,\sigma(i)}$$

$$\begin{aligned} pf(A) &= \frac{1}{2^n n!} \sum_{\sigma \in S_{2n}} \operatorname{sgn}(\sigma) \prod_{i=1}^{n} a_{\sigma(2i-1),\sigma(2i)} \\ &= \sum_{M \in \mathrm{PM}(n)} \operatorname{sgn}(M) \prod_{(i,j) \in M} a_{i,j} \end{aligned}$$

11.5.2. *BEST Theorem.*

Count directed Eulerian cycles. Number of OST given by Kirchoff's Theorem (remove r/c with root) $\#\mathrm{OST}(G, r) \cdot \prod_v (d_v - 1)!$

11.5.3. *Primitive Roots.*

Only exists when n is $2, 4, p^k, 2p^k$, where p odd prime. Assume n prime. Number of primitive roots $\phi(\phi(n))$ Let g be primitive root. All primitive roots are of the form g^k where $k, \phi(p)$ are coprime.
k-roots: $g^{i \cdot \phi(n)/k}$ for $0 \le i < k$

11.5.4. *Sum of primes.*

For any multiplicative f:

$$S(n, p) = S(n, p-1) - f(p) \cdot (S(n/p, p-1) - S(p-1, p-1))$$

11.5.5. *Floor.*

$$\lfloor \lfloor x/y \rfloor / z \rfloor = \lfloor x/(yz) \rfloor$$

$$x\%y = x - y \lfloor x/y \rfloor$$

Practice Contest Checklist

- How many operations per second? Compare to local machine.
- What is the stack size?
- How to use printf/scanf with long long/long double?
- Are `__int128` and `__float128` available?
- Does MLE give RTE or MLE as a verdict? What about stack overflow?
- What is `RAND_MAX`?
- How does the judge handle extra spaces (or missing newlines) in the output?
- Look at documentation for programming languages.
- Try different programming languages: C++, Java and Python.
- Try the submit script.
- Try local programs: i?python[23], factor.
- Try submitting with `assert(false)` and `assert(true)`.
- Return-value from `main`.
- Look for directory with sample test cases.
- Make sure printing works.
- Remove this page from the notebook.