AdMU Progvar

Team Notebook

26/09/2019

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. Mergesort Tree.

2.3. Segment Tree.

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

```
struct segtree { -----------------------------------------//0c2
- int i, j, val; -----------------------------------------//0c3
- segtree *l, *r; ----------------------------------------//0c4
- segtree(int *ar, int _i, int _j) : i(_i), j(_j) { ------//0c5
--- if (i == j) { ----------------------------------------//0c6
----- val = ar[i]; ---------------------------------------//0c7
----- l = r = NULL; --------------------------------------//0c8
--- } else { ---------------------------------------------//0c9
----- int k = (i+j) >> 1; --------------------------------//0ca
----- l = new segtree(ar, i, k); -------------------------//0cb
----- r = new segtree(ar, k+1, j); -----------------------//0cc
----- val = l->val + r->val; -----------------------------//0cd
--- } ----------------------------------------------------//0ce
- } ------------------------------------------------------//0cf
- void update(int _i, int _val) { ------------------------//0d0
--- if (i == _i and _i == j) { ---------------------------//0d1
----- val = _val; ----------------------------------------//0d2
--- } else if (_i < i or j < _i) { -----------------------//0d3
----- // do nothing --------------------------------------//0d4
--- } else { ---------------------------------------------//0d5
----- l->update(_i, _val); -------------------------------//0d6
----- r->update(_i, _val); -------------------------------//0d7
----- val = l->val + r->val; -----------------------------//0d8
--- } ----------------------------------------------------//0d9
- } ------------------------------------------------------//0da
- int query(int _i, int _j) { ----------------------------//0db
--- if (_i <= i and j <= _j) { ---------------------------//0dc
----- return val; ----------------------------------------//0dd
--- } else if (_j < i or j < _i) { -----------------------//0de
----- return 0; ------------------------------------------//0df
--- } else { ---------------------------------------------//0e0
----- return l->query(_i, _j) + r->query(_i, _j); --------//0e1
--- } ----------------------------------------------------//0e2
- } ------------------------------------------------------//0e3
}; -------------------------------------------------------//0e4
```

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

```
struct segtree { -----------------------------------------//049
- int n; -------------------------------------------------//04a
- int *vals; ---------------------------------------------//04b
- segtree(int *ar, int n) { ------------------------------//04c
--- this->n = n; -----------------------------------------//04d
--- vals = new int[2*n]; ---------------------------------//04e
--- for (int i = 0; i < n; ++i) --------------------------//04f
----- vals[n+i] = ar[i]; ---------------------------------//050
--- for (int i = n-1; i > 0; --i) ------------------------//051
----- vals[i] = vals[i<<1] + vals[i<<1|1]; ---------------//052
- } ------------------------------------------------------//053
- void update(int i, int v) { ----------------------------//054
--- for (vals[i += n] = v; i > 1; i >>= 1) ---------------//055
----- vals[i>>1] = vals[i] + vals[i^1]; ------------------//056
- } ------------------------------------------------------//057
- int query(int l, int r) { ------------------------------//058
--- r++;    // without this, the range is [l,r) ----------//059
--- int res = 0; -----------------------------------------//05a
--- for (l += n, r += n; l < r; l >>= 1, r >>= 1) { ------//05b
----- if (l&1)  res += vals[l++]; ------------------------//05c
----- if (r&1)  res += vals[--r]; ------------------------//05d
--- } ----------------------------------------------------//05e
--- return res; ------------------------------------------//05f
- } ------------------------------------------------------//060
}; -------------------------------------------------------//061
```

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

```
struct segtree { -----------------------------------------//062
- int i, j, val, temp_val = 0; ---------------------------//063
- segtree *l, *r; ----------------------------------------//064
- segtree(int *ar, int _i, int _j) : i(_i), j(_j) { ------//065
--- if (i == j) { ----------------------------------------//066
----- val = ar[i]; ---------------------------------------//067
----- l = r = NULL; --------------------------------------//068
--- } else { ---------------------------------------------//069
----- int k = (i + j) >> 1; ------------------------------//06a
----- l = new segtree(ar, i, k); -------------------------//06b
----- r = new segtree(ar, k+1, j); -----------------------//06c
----- val = l->val + r->val; -----------------------------//06d
--- } ----------------------------------------------------//06e
- } ------------------------------------------------------//06f
- void visit() { -----------------------------------------//070
--- if (temp_val) { --------------------------------------//071
----- val += (j-i+1) * temp_val; -------------------------//072
----- if (l) { -------------------------------------------//073
------- l->temp_val += temp_val; -------------------------//074
------- r->temp_val += temp_val; -------------------------//075
----- } --------------------------------------------------//076
----- temp_val = 0; --------------------------------------//077
--- } ----------------------------------------------------//078
- } ------------------------------------------------------//079
- void increase(int _i, int _j, int _inc) { --------------//07a
--- visit(); ---------------------------------------------//07b
--- if (_i <= i && j <= _j) { ----------------------------//07c
----- temp_val += _inc; ----------------------------------//07d
----- visit(); -------------------------------------------//07e
--- } else if (_j < i or j < _i) { -----------------------//07f
----- // do nothing --------------------------------------//080
--- } else { ---------------------------------------------//081
----- l->increase(_i, _j, _inc); -------------------------//082
----- r->increase(_i, _j, _inc); -------------------------//083
----- val = l->val + r->val; -----------------------------//084
--- } ----------------------------------------------------//085
- } ------------------------------------------------------//086
- int query(int _i, int _j) { ----------------------------//087
--- visit(); ---------------------------------------------//088
--- if (_i <= i and j <= _j) { ---------------------------//089
----- return val; ----------------------------------------//08a
--- } else if (_j < i || j < _i) { -----------------------//08b
----- return 0; ------------------------------------------//08c
--- } else { ---------------------------------------------//08d
----- return l->query(_i, _j) + r->query(_i, _j); --------//08e
--- } ----------------------------------------------------//08f
- } ------------------------------------------------------//090
}; -------------------------------------------------------//091
```

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

```
struct node {  int l, r, lid, rid, val;  }; --------------//092
struct segtree { -----------------------------------------//093
- node *nodes; -------------------------------------------//094
- int n, node_cnt = 0; -----------------------------------//095
- segtree(int n, int capacity) { -------------------------//096
--- this->n = n; -----------------------------------------//097
--- nodes = new node[capacity]; --------------------------//098
- } ------------------------------------------------------//099
- int build (int *ar, int l, int r) { --------------------//09a
--- if (l > r)  return -1; -------------------------------//09b
--- int id = node_cnt++; ---------------------------------//09c
--- nodes[id].l = l; -------------------------------------//09d
--- nodes[id].r = r; -------------------------------------//09e
--- if (l == r) { ----------------------------------------//09f
----- nodes[id].lid = -1; --------------------------------//0a0
----- nodes[id].rid = -1; --------------------------------//0a1
----- nodes[id].val = ar[l]; -----------------------------//0a2
--- } else { ---------------------------------------------//0a3
----- int m = (l + r) / 2; -------------------------------//0a4
----- nodes[id].lid = build(ar, l, m); -------------------//0a5
----- nodes[id].rid = build(ar, m+1, r); -----------------//0a6
----- nodes[id].val = nodes[nodes[id].lid].val + ----------//0a7
--------------------- nodes[nodes[id].rid].val; ----------//0a8
--- } ----------------------------------------------------//0a9
--- return id; -------------------------------------------//0aa
- } ------------------------------------------------------//0ab
- int update(int id, int idx, int delta) { ---------------//0ac
--- if (id == -1) ----------------------------------------//0ad
----- return -1; -----------------------------------------//0ae
--- if (idx < nodes[id].l or nodes[id].r < idx) ----------//0af
----- return id; -----------------------------------------//0b0
--- int nid = node_cnt++; --------------------------------//0b1
--- nodes[nid].l = nodes[id].l; --------------------------//0b2
--- nodes[nid].r = nodes[id].r; --------------------------//0b3
--- nodes[nid].lid = update(nodes[id].lid, idx, delta); --//0b4
--- nodes[nid].rid = update(nodes[id].rid, idx, delta); --//0b5
--- nodes[nid].val = nodes[id].val + delta; --------------//0b6
--- return nid; ------------------------------------------//0b7
- } ------------------------------------------------------//0b8
- int query(int id, int l, int r) { ----------------------//0b9
--- if (r < nodes[id].l or nodes[id].r < l) --------------//0ba
----- return 0; ------------------------------------------//0bb
--- if (l <= nodes[id].l and nodes[id].r <= r) -----------//0bc
----- return nodes[id].val; ------------------------------//0bd
--- return query(nodes[id].lid, l, r) + ------------------//0be
---------- query(nodes[id].rid, l, r); -------------------//0bf
-- } -----------------------------------------------------//0c0
 }; ------------------------------------------------------//0c1
```

2.4. **Sparse Table.**

2.5. **Sqrt Decomposition.**

2.6. **Treap.**

2.6.1. *Explicit Treap.*

2.6.2. *Implicit Treap.*

```
struct cartree { //0e5
- typedef struct _Node { //0e6
--- int node_val, subtree_val, delta, prio, size; //0e7
--- _Node *l, *r; //0e8
--- _Node(int val) : node_val(val), subtree_val(val), //0e9
------- delta(0), prio((rand()<<16)^rand()), size(1), //0ea
------- l(NULL), r(NULL) {} //0eb
--- ~_Node() { delete l; delete r; } //0ec
- } *Node; //0ed
- int get_subtree_val(Node v) { //0ee
--- return v ? v->subtree_val : 0; } //0ef
- int get_size(Node v) { return v ? v->size : 0; } //0f0
- void apply_delta(Node v, int delta) { //0f1
--- if (!v) return; //0f2
--- v->delta += delta; //0f3
--- v->node_val += delta; //0f4
--- v->subtree_val += delta * get_size(v); //0f5
- } //0f6
- void push_delta(Node v) { //0f7
--- if (!v) return; //0f8
--- apply_delta(v->l, v->delta); //0f9
--- apply_delta(v->r, v->delta); //0fa
--- v->delta = 0; //0fb
- } //0fc
- void update(Node v) { //0fd
--- if (!v) return; //0fe
--- v->subtree_val = get_subtree_val(v->l) + v->node_val -//0ff
----------------- + get_subtree_val(v->r); //100
--- v->size = get_size(v->l) + 1 + get_size(v->r); //101
- } //102
- Node merge(Node l, Node r) { //103
--- push_delta(l);    push_delta(r); //104
--- if (!l || !r)   return l ? l : r; //105
--- if (l->size <= r->size) { //106
----- l->r = merge(l->r, r); //107
----- update(l); //108
----- return l; //109
--- } else { //10a
----- r->l = merge(l, r->l); //10b
----- update(r); //10c
----- return r; //10d
--- } //10e
- } //10f
- void split(Node v, int key, Node &l, Node &r) { //110
--- push_delta(v); //111
--- l = r = NULL; //112
--- if (!v)     return; //113
--- if (key <= get_size(v->l)) { //114
----- split(v->l, key, l, v->l); //115
----- r = v; //116
--- } else { //117
----- split(v->r, key - get_size(v->l) - 1, v->r, r); //118
----- l = v; //119
--- } //11a
--- update(v); //11b
- } //11c
- Node root; //11d
public: //11e
- cartree() : root(NULL) {} //11f
- ~cartree() { delete root; } //120
- int get(Node v, int key) { //121
--- push_delta(v); //122
--- if (key < get_size(v->l)) //123
----- return get(v->l, key); //124
--- else if (key > get_size(v->l)) //125
----- return get(v->r, key - get_size(v->l) - 1); //126
--- return v->node_val;  } //127
- int get(int key) { return get(root, key); } //128
- void insert(Node item, int key) { //129
--- Node l, r; //12a
--- split(root, key, l, r); //12b
--- root = merge(merge(l, item), r); //12c
- } //12d
- void insert(int key, int val) { //12e
--- insert(new _Node(val), key); //12f
- } //130
- void erase(int key) { //131
--- Node l, m, r; //132
--- split(root, key + 1, m, r); //133
--- split(m, key, l, m); //134
--- delete m; //135
--- root = merge(l, r); //136
- } //137
- int query(int a, int b) { //138
--- Node l1, r1; //139
--- split(root, b+1, l1, r1); //13a
--- Node l2, r2; //13b
--- split(l1, a, l2, r2); //13c
--- int res = get_subtree_val(r2); //13d
--- l1 = merge(l2, r2); //13e
--- root = merge(l1, r1); //13f
--- return res; //140
- } //141
- int update(int a, int b, int delta) { //142
--- Node l1, r1; //143
--- split(root, b+1, l1, r1); //144
--- Node l2, r2; //145
--- split(l1, a, l2, r2); //146
--- apply_delta(r2, delta); //147
--- l1 = merge(l2, r2); //148
--- root = merge(l1, r1); //149
- } //14a
- int size() { return get_size(root); }  }; //14b
```

2.6.3. *Persistent Treap.*

2.7. **Ordered Statistics Tree.**

2.8. **Union Find.**

```
struct union_find { //14c
- vi p; union_find(int n) : p(n, -1) { } //14d
- int find(int x) { return p[x] < 0 ? x : p[x] = find(p[x]); } //14e
- bool unite(int x, int y) { //14f
--- int xp = find(x), yp = find(y); //150
--- if (xp == yp) return false; //151
--- if (p[xp] > p[yp]) swap(xp,yp); //152
--- p[xp] += p[yp], p[yp] = xp; //153
--- return true; //154
- } //155
- int size(int x) { return -p[find(x)]; } //156
}; //157
```

3. Graphs

Using adjacency list:

```
struct graph { //214
- int n; //215
- vii *adj; //216
- int *dist; //217
- graph(int n) { //218
--- this->n = n; //219
--- adj = new vii[n]; //21a
--- dist = new int[n]; //21b
- } //21c
- void add_edge(int u, int v, int w) { //21d
--- adj[u].push_back({v, w}); //21e
--- /*adj[v].push_back({u, w});*/ //21f
- } //220
}; //221
```

Using adjacency matrix:

```
struct graph { //222
- int n; //223
- int **mat; //224
- graph(int n) { //225
--- this->n = n; //226
--- mat = new int*[n]; //227
--- for (int i = 0; i < n; ++i) { //228
----- mat[i] = new int[n]; //229
----- for (int j = 0; j < n; ++j) //22a
------- mat[i][j] = INF; //22b
----- mat[i][i] = 0; //22c
--- } //22d
- } //22e
- void add_edge(int u, int v, int w) { //22f
--- mat[u][v] = std::min(mat[u][v], w); //230
- /*mat[v][u] = std::min(mat[v][u], w);*/ //231
- } //232
}; //233
```

Using edge list:

```
struct edge { //234
- int u, v, w; //235
- edge(int u, int v, int w) : u(u), v(v), w(w) {} //236
- const bool operator <(const edge &other) const { //237
--- return w < other.w; //238
- } //239
```

```
}; -------------------------------------------------------//23a
struct graph { --------------------------------------------//23b
- int n; --------------------------------------------------//23c
- std::vector<edge> edges; --------------------------------//23d
- graph(int n) : n(n) {} ----------------------------------//23e
- void add_edge(int u, int v, int w) { --------------------//23f
--- edges.push_back(edge(u, v, w)); -----------------------//240
- } -------------------------------------------------------//241
}; --------------------------------------------------------//242
```

3.1. Single-Source Shortest Paths.

3.1.1. *Dijkstra.*

```
#include "graph_template_adjlist.cpp" ---------------------//16c
// insert inside graph; needs n, dist[], and adj[] --------//16d
void dijkstra(int s) { ------------------------------------//16e
- for (int u = 0; u < n; ++u) -----------------------------//16f
--- dist[u] = INF; ----------------------------------------//170
- dist[s] = 0; --------------------------------------------//171
- std::priority_queue<ii, vii, std::greater<ii> > pq; -----//172
- pq.push({0, s}); ----------------------------------------//173
- while (!pq.empty()) { -----------------------------------//174
--- int u = pq.top().second; ------------------------------//175
--- int d = pq.top().first; -------------------------------//176
--- pq.pop(); ---------------------------------------------//177
--- if (dist[u] < d) --------------------------------------//178
----- continue; -------------------------------------------//179
--- dist[u] = d; ------------------------------------------//17a
--- for (auto &e : adj[u]) { ------------------------------//17b
----- int v = e.first; ------------------------------------//17c
----- int w = e.second; -----------------------------------//17d
----- if (dist[v] > dist[u] + w) { ------------------------//17e
------- dist[v] = dist[u] + w; ----------------------------//17f
------- pq.push({dist[v], v}); ----------------------------//180
----- } ---------------------------------------------------//181
--- } -----------------------------------------------------//182
- } -------------------------------------------------------//183
} ---------------------------------------------------------//184
```

3.1.2. *Bellman-Ford.*

```
#include "graph_template_adjlist.cpp" ---------------------//158
// insert inside graph; needs n, dist[], and adj[] --------//159
void bellman_ford(int s) { --------------------------------//15a
- for (int u = 0; u < n; ++u) -----------------------------//15b
--- dist[u] = INF; ----------------------------------------//15c
- dist[s] = 0; --------------------------------------------//15d
- for (int i = 0; i < n-1; ++i) ---------------------------//15e
--- for (int u = 0; u < n; ++u) ---------------------------//15f
----- for (auto &e : adj[u]) ------------------------------//160
------- if (dist[u] + e.second < dist[e.first]) ----------//161
--------- dist[e.first] = dist[u] + e.second; -------------//162
} ---------------------------------------------------------//163
// you can call this after running bellman_ford() --------//164
bool has_neg_cycle() { ------------------------------------//165
- for (int u = 0; u < n; ++u) -----------------------------//166
--- for (auto &e : adj[u]) --------------------------------//167
----- if (dist[e.first] > dist[u] + e.second) ------------//168
------- return true; --------------------------------------//169
- return false; -------------------------------------------//16a
} ---------------------------------------------------------//16b
```

3.2. All-Pairs Shortest Paths.

3.2.1. *Floyd-Washall.*

```
#include "graph_template_adjmat.cpp" ----------------------//20b
// insert inside graph; needs n and mat[][] ---------------//20c
void floyd_warshall() { -----------------------------------//20d
- for (int k = 0; k < n; ++k) -----------------------------//20e
--- for (int i = 0; i < n; ++i) ---------------------------//20f
----- for (int j = 0; j < n; ++j) -------------------------//210
------- if (mat[i][k] + mat[k][j] < mat[i][j]) -----------//211
--------- mat[i][j] = mat[i][k] + mat[k][j]; --------------//212
} ---------------------------------------------------------//213
```

3.3. Strongly Connected Components.

3.3.1. *Kosaraju.*

```
struct kosaraju_graph { -----------------------------------//28c
- int n; --------------------------------------------------//28d
- int *vis; -----------------------------------------------//28e
- vi **adj; -----------------------------------------------//28f
- std::vector<vi> sccs; -----------------------------------//290
- kosaraju_graph(int n) { ---------------------------------//291
--- this->n = n; ------------------------------------------//292
--- vis = new int[n]; -------------------------------------//293
--- adj = new vi*[2]; -------------------------------------//294
--- for (int dir = 0; dir < 2; ++dir) ---------------------//295
----- adj[dir] = new vi[n]; -------------------------------//296
- } -------------------------------------------------------//297
- void add_edge(int u, int v) { ---------------------------//298
--- adj[0][u].push_back(v); -------------------------------//299
--- adj[1][v].push_back(u); -------------------------------//29a
- } -------------------------------------------------------//29b
- void dfs(int u, int p, int dir, vi &topo) { -------------//29c
--- vis[u] = 1; -------------------------------------------//29d
--- for (int v : adj[dir][u]) -----------------------------//29e
----- if (!vis[v] && v != p) ------------------------------//29f
------- dfs(v, u, dir, topo); -----------------------------//2a0
--- topo.push_back(u); ------------------------------------//2a1
- } -------------------------------------------------------//2a2
- void kosaraju() { ---------------------------------------//2a3
--- vi topo; ----------------------------------------------//2a4
--- for (int u = 0; u < n; ++u)  vis[u] = 0; --------------//2a5
--- for (int u = 0; u < n; ++u) ---------------------------//2a6
----- if (!vis[u]) ----------------------------------------//2a7
------- dfs(u, -1, 0, topo); ------------------------------//2a8
--- for (int u = 0; u < n; ++u)  vis[u] = 0; --------------//2a9
--- for (int i = n-1; i >= 0; --i) { ----------------------//2aa
----- if (!vis[topo[i]]) { --------------------------------//2ab
------- sccs.push_back({}); -------------------------------//2ac
------- dfs(topo[i], -1, 1, sccs.back()); -----------------//2ad
----- } ---------------------------------------------------//2ae
--- } -----------------------------------------------------//2af
- } -------------------------------------------------------//2b0
}; --------------------------------------------------------//2b1
```

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 { -------------------------------------//1d3
- int n, s, t, *par, **c, **f; ----------------------------//1d4
- vi *adj; ------------------------------------------------//1d5
- flow_network(int n, int s, int t) : n(n), s(s), t(t) { -//1d6
--- adj = new std::vector<int>[n]; ------------------------//1d7
--- par = new int[n]; -------------------------------------//1d8
--- c = new int*[n]; --------------------------------------//1d9
--- f = new int*[n]; --------------------------------------//1da
--- for (int i = 0; i < n; ++i) { -------------------------//1db
----- c[i] = new int[n]; ----------------------------------//1dc
----- f[i] = new int[n]; ----------------------------------//1dd
----- for (int j = 0; j < n; ++j) -------------------------//1de
------- c[i][j] = f[i][j] = 0; ----------------------------//1df
--- } -----------------------------------------------------//1e0
- } -------------------------------------------------------//1e1
- void add_edge(int u, int v, int w) { --------------------//1e2
--- adj[u].push_back(v); ----------------------------------//1e3
--- adj[v].push_back(u); ----------------------------------//1e4
--- c[u][v] += w; -----------------------------------------//1e5
- } -------------------------------------------------------//1e6
- int res(int i, int j) { return c[i][j] - f[i][j]; } ----//1e7
- bool bfs() { --------------------------------------------//1e8
--- std::queue<int> q; ------------------------------------//1e9
--- q.push(this->s); --------------------------------------//1ea
--- while (!q.empty()) { ----------------------------------//1eb
----- int u = q.front();  q.pop(); ------------------------//1ec
----- for (int v : adj[u]) { ------------------------------//1ed
------- if (res(u, v) > 0 and par[v] == -1) { ------------//1ee
--------- par[v] = u; -------------------------------------//1ef
--------- if (v == this->t) -------------------------------//1f0
----------- return true; ----------------------------------//1f1
--------- q.push(v); --------------------------------------//1f2
------- } -------------------------------------------------//1f3
----- } ---------------------------------------------------//1f4
--- } -----------------------------------------------------//1f5
```

```
--- return false; ----------------------------------------//1f6
- } ------------------------------------------------------//1f7
- bool aug_path() { --------------------------------------//1f8
--- for (int u = 0; u < n; ++u) --------------------------//1f9
----- par[u] = -1; ---------------------------------------//1fa
--- par[s] = s; ------------------------------------------//1fb
--- return bfs(); ----------------------------------------//1fc
- } ------------------------------------------------------//1fd
- int calc_max_flow() { ----------------------------------//1fe
--- int ans = 0; -----------------------------------------//1ff
--- while (aug_path()) { ---------------------------------//200
----- int flow = INF; ------------------------------------//201
----- for (int u = t; u != s; u = par[u]) ----------------//202
------- flow = std::min(flow, res(par[u], u)); -----------//203
----- for (int u = t; u != s; u = par[u]) ----------------//204
------- f[par[u]][u] += flow, f[u][par[u]] -= flow; ------//205
----- ans += flow; ---------------------------------------//206
--- } ----------------------------------------------------//207
--- return ans; ------------------------------------------//208
- } ------------------------------------------------------//209
}; -------------------------------------------------------//20a
```

3.10.2. *Dinic.*

```
struct flow_network { ------------------------------------//185
- int n, s, t, *adj_ptr, *dist, *par, **c, **f; ----------//186
- vi *adj; -----------------------------------------------//187
- flow_network(int n, int s, int t) : n(n), s(s), t(t) { -//188
--- adj = new std::vector<int>[n]; -----------------------//189
--- adj_ptr = new int[n]; --------------------------------//18a
--- dist = new int[n]; -----------------------------------//18b
--- par = new int[n]; ------------------------------------//18c
--- c = new int*[n]; -------------------------------------//18d
--- f = new int*[n]; -------------------------------------//18e
--- for (int i = 0; i < n; ++i) { ------------------------//18f
----- c[i] = new int[n]; ---------------------------------//190
----- f[i] = new int[n]; ---------------------------------//191
----- for (int j = 0; j < n; ++j) ------------------------//192
------- c[i][j] = f[i][j] = 0; ---------------------------//193
--- } ----------------------------------------------------//194
- } ------------------------------------------------------//195
- void add_edge(int u, int v, int w) { -------------------//196
--- adj[u].push_back(v); ---------------------------------//197
--- adj[v].push_back(u); ---------------------------------//198
--- c[u][v] += w; ----------------------------------------//199
- } ------------------------------------------------------//19a
- int res(int i, int j) { return c[i][j] - f[i][j]; } ----//19b
- void reset(int *ar, int val) { -------------------------//19c
--- for (int i = 0; i < n; ++i) --------------------------//19d
----- ar[i] = val; ---------------------------------------//19e
- } ------------------------------------------------------//19f
- bool make_level_graph() { ------------------------------//1a0
--- reset(dist, -1); -------------------------------------//1a1
--- std::queue<int> q; -----------------------------------//1a2
--- q.push(s); -------------------------------------------//1a3
--- dist[s] = 0; -----------------------------------------//1a4
--- while (!q.empty()) { ---------------------------------//1a5
----- int u = q.front();  q.pop(); -----------------------//1a6
----- for (int v : adj[u]) { -----------------------------//1a7
------- if (res(u, v) > 0 and dist[v] == -1) { -----------//1a8
--------- dist[v] = dist[u] + 1; -------------------------//1a9
--------- q.push(v); -------------------------------------//1aa
------- } ------------------------------------------------//1ab
----- } --------------------------------------------------//1ac
--- } ----------------------------------------------------//1ad
--- return dist[t] != -1; --------------------------------//1ae
- } ------------------------------------------------------//1af
- bool next(int u, int v) { ------------------------------//1b0
--- return dist[v] == dist[u] + 1; -----------------------//1b1
- } ------------------------------------------------------//1b2
- bool dfs(int u) { --------------------------------------//1b3
--- if (u == t)   return true; ---------------------------//1b4
--- for (int &i = adj_ptr[u]; i < adj[u].size(); ++i) { --//1b5
----- int v = adj[u][i]; ---------------------------------//1b6
----- if (next(u, v) and res(u, v) > 0 and dfs(v)) { -----//1b7
------- par[v] = u; --------------------------------------//1b8
------- return true; -------------------------------------//1b9
----- } --------------------------------------------------//1ba
--- } ----------------------------------------------------//1bb
--- dist[u] = -1; ----------------------------------------//1bc
--- return false; ----------------------------------------//1bd
- } ------------------------------------------------------//1be
- bool aug_path() { --------------------------------------//1bf
--- reset(par, -1); --------------------------------------//1c0
--- par[s] = s; ------------------------------------------//1c1
--- return dfs(s);  } ------------------------------------//1c2
- int calc_max_flow() { ----------------------------------//1c3
--- int ans = 0; -----------------------------------------//1c4
--- while (make_level_graph()) { -------------------------//1c5
----- reset(adj_ptr, 0); ---------------------------------//1c6
----- while (aug_path()) { -------------------------------//1c7
------- int flow = INF; ----------------------------------//1c8
------- for (int u = t; u != s; u = par[u]) --------------//1c9
--------- flow = std::min(flow, res(par[u], u)); ---------//1ca
------- for (int u = t; u != s; u = par[u]) --------------//1cb
--------- f[par[u]][u] += flow, f[u][par[u]] -= flow; ----//1cc
------- ans += flow; -------------------------------------//1cd
----- } --------------------------------------------------//1ce
--- } ----------------------------------------------------//1cf
--- return ans; ------------------------------------------//1d0
- } ------------------------------------------------------//1d1
}; -------------------------------------------------------//1d2
```

3.11. **All-pairs Maximum Flow.**

3.11.1. *Gomory-Hu.*

3.12. **Heavy Light Decomposition.**

```
#include "segment_tree.cpp" ------------------------------//243
struct heavy_light_tree { --------------------------------//244
- int n; -------------------------------------------------//245
- std::vector<int> *adj; ---------------------------------//246
- segtree *segment_tree; ---------------------------------//247
- int *par, *heavy, *dep, *path_root, *pos; --------------//248
- heavy_light_tree(int n) { ------------------------------//249
--- this->n = n; -----------------------------------------//24a
--- this->adj = new std::vector<int>[n]; -----------------//24b
--- segment_tree = new segtree(0, n-1); ------------------//24c
--- par = new int[n]; ------------------------------------//24d
--- heavy = new int[n]; ----------------------------------//24e
--- dep = new int[n]; ------------------------------------//24f
--- path_root = new int[n]; ------------------------------//250
--- pos = new int[n]; ------------------------------------//251
- } ------------------------------------------------------//252
- void add_edge(int u, int v) { --------------------------//253
--- adj[u].push_back(v); ---------------------------------//254
--- adj[v].push_back(u); ---------------------------------//255
- } ------------------------------------------------------//256
- void build(int root) { ---------------------------------//257
--- for (int u = 0; u < n; ++u) --------------------------//258
----- heavy[u] = -1; -------------------------------------//259
--- par[root] = root; ------------------------------------//25a
--- dep[root] = 0; ---------------------------------------//25b
--- dfs(root); -------------------------------------------//25c
--- for (int u = 0, p = 0; u < n; ++u) { -----------------//25d
----- if (par[u] == -1 or heavy[par[u]] != u) { ----------//25e
------- for (int v = u; v != -1; v = heavy[v]) { ---------//25f
--------- path_root[v] = u; ------------------------------//260
--------- pos[v] = p++; ----------------------------------//261
------- } ------------------------------------------------//262
----- } --------------------------------------------------//263
--- } ----------------------------------------------------//264
- } ------------------------------------------------------//265
- int dfs(int u) { ---------------------------------------//266
--- int sz = 1; ------------------------------------------//267
--- int max_subtree_sz = 0; ------------------------------//268
--- for (int v : adj[u]) { -------------------------------//269
----- if (v != par[u]) { ---------------------------------//26a
------- par[v] = u; --------------------------------------//26b
------- dep[v] = dep[u] + 1; -----------------------------//26c
------- int subtree_sz = dfs(v); -------------------------//26d
------- if (max_subtree_sz < subtree_sz) { ---------------//26e
--------- max_subtree_sz = subtree_sz; -------------------//26f
--------- heavy[u] = v; ----------------------------------//270
------- } ------------------------------------------------//271
------- sz += subtree_sz; --------------------------------//272
----- } --------------------------------------------------//273
--- } ----------------------------------------------------//274
--- return sz; -------------------------------------------//275
- } ------------------------------------------------------//276
- int query(int u, int v) { ------------------------------//277
--- int res = 0; -----------------------------------------//278
--- while (path_root[u] != path_root[v]) { ---------------//279
----- if (dep[path_root[u]] > dep[path_root[v]]) --------//27a
------- std::swap(u, v); ---------------------------------//27b
----- res += segment_tree->sum(pos[path_root[v]], pos[v]);//27c
----- v = par[path_root[v]]; -----------------------------//27d
--- } ----------------------------------------------------//27e
--- res += segment_tree->sum(pos[u], pos[v]); ------------//27f
--- return res; ------------------------------------------//280
```

```
- } ---------------------------------------------------//281
- void update(int u, int v, int c) { --------------------//282
--- for (; path_root[u] != path_root[v]; ----------------//283
---------- v = par[path_root[v]]) { ---------------------//284
----- if (dep[path_root[u]] > dep[path_root[v]]) ---------//285
------- std::swap(u, v); ---------------------------------//286
----- segment_tree->increase(pos[path_root[v]], pos[v], c);
--- } ---------------------------------------------------//288
--- segment_tree->increase(pos[u], pos[v], c); ----------//289
- } -----------------------------------------------------//28a
}; ------------------------------------------------------//28b
```

3.13. **Centroid Decomposition.**

3.14. **Least Common Ancestor.**

3.14.1. *Binary Lifting.*

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; --------------------------//2b2
struct hasher { ------------------------------------------//2b3
- int n; --------------------------------------------------//2b4
- std::vector<ll> *p_pow; --------------------------------//2b5
- std::vector<ll> *h_ans; --------------------------------//2b6
- hash(vi &s, vi primes) { -------------------------------//2b7
--- n = primes.size(); -----------------------------------//2b8
--- p_pow = new std::vector<ll>[n]; ---------------------//2b9
--- h_ans = new std::vector<ll>[n]; ---------------------//2ba
--- for (int i = 0; i < n; ++i) { -----------------------//2bb
----- p_pow[i] = std::vector<ll>(MAXN); -----------------//2bc
----- p_pow[i][0] = 1; ----------------------------------//2bd
----- for (int j = 0; j+1 < MAXN; ++j) ------------------//2be
------- p_pow[i][j+1] = (p_pow[i][j] * primes[i]) % MOD; -//2bf
----- h_ans[i] = std::vector<ll>(MAXN); -----------------//2c0
----- h_ans[i][0] = 0; ----------------------------------//2c1
----- for (int j = 0; j < s.size(); ++j) ----------------//2c2
------- h_ans[i][j+1] = (h_ans[i][j] + ------------------//2c3
------------------------ s[j] * p_pow[i][j]) % MOD; -----//2c4
--- } ---------------------------------------------------//2c5
- } -----------------------------------------------------//2c6
}; ------------------------------------------------------//2c7
```

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.**

6.8. **Modular Multiplicative Inverse.**

6.9. **Chinese Remainder Theorem.**

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 $L(x) = \sum_{j=0}^{k} y_j \prod_{\substack{0 \le m \le k \\ m \ne j}} \frac{x - x_m}{x_j - x_m}$
- **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.