L^3

Team Notebook

25/09/2019

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. *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. *Range Queries.*

2.2. Mergesort Tree.

2.3. Segment Tree.

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

```
struct segtree {
- int i, j, val;
- segtree *l, *r;
- segtree(int *ar, int _i, int _j) : i(_i), j(_j) {
--- if (i == j) {
----- val = ar[i];
----- l = r = NULL;
--- } else {
----- int k = (i+j) >> 1;
----- l = new segtree(ar, i, k);
----- r = new segtree(ar, k+1, j);
----- val = l->val + r->val; } }
- void update(int _i, int _val) {
--- if (i == _i and _i == j) {
----- val = _val;
--- } else if (_i < i or j < _i) {
----- // do nothing
--- } else {
----- l->update(_i, _val);
----- r->update(_i, _val);
----- val = l->val + r->val; } }
- int query(int _i, int _j) {
--- if (_i <= i and j <= _j) {
----- return val;
--- } else if (_j < i or j < _i) {
----- return 0;
--- } else {
----- return l->query(_i, _j) + r->query(_i, _j); } } };
```

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

```
struct segtree { -------------------------------------------//02c
- int n; --------------------------------------------------//02d
- int *vals; ----------------------------------------------//02e
- segtree(int *ar, int n) { -------------------------------//02f
--- this->n = n; ------------------------------------------//030
--- vals = new int[2*n]; ----------------------------------//031
--- for (int i = 0; i < n; ++i) ---------------------------//032
----- vals[n+i] = ar[i]; ----------------------------------//033
--- for (int i = n-1; i > 0; --i) -------------------------//034
----- vals[i] = vals[i<<1] + vals[i<<1|1]; } --------------//035
- void update(int i, int v) { -----------------------------//036
--- for (vals[i += n] = v; i > 1; i >>= 1) ----------------//037
----- vals[i>>1] = vals[i] + vals[i^1]; } -----------------//038
- int query(int l, int r) { -------------------------------//039
--- r++;    // without this, the range is [l,r) -----------//03a
--- int res = 0; ------------------------------------------//03b
--- for (l += n, r += n; l < r; l >>= 1, r >>= 1) { -------//03c
----- if (l&1)  res += vals[l++]; -------------------------//03d
----- if (r&1)  res += vals[--r]; } -----------------------//03e
--- return res;  }  }; ------------------------------------//03f
```

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

```
struct segtree { -------------------------------------------//040
- int i, j, val, temp_val = 0; -----------------------------//041
- segtree *l, *r; ------------------------------------------//042
- segtree(int *ar, int _i, int _j) : i(_i), j(_j) { --------//043
--- if (i == j) { ------------------------------------------//044
----- val = ar[i]; -----------------------------------------//045
----- l = r = NULL; ----------------------------------------//046
--- } else { -----------------------------------------------//047
----- int k = (i + j) >> 1; --------------------------------//048
----- l = new segtree(ar, i, k); ---------------------------//049
----- r = new segtree(ar, k+1, j); -------------------------//04a
----- val = l->val + r->val;  }  } -------------------------//04b
- void visit() { -------------------------------------------//04c
--- if (temp_val) { ----------------------------------------//04d
----- val += (j-i+1) * temp_val; ---------------------------//04e
----- if (l) { ---------------------------------------------//04f
------- l->temp_val += temp_val; ---------------------------//050
------- r->temp_val += temp_val;  } ------------------------//051
----- temp_val = 0;  }  } ----------------------------------//052
- void increase(int _i, int _j, int _inc) { ----------------//053
--- visit(); -----------------------------------------------//054
--- if (_i <= i && j <= _j) { ------------------------------//055
----- temp_val += _inc; ------------------------------------//056
----- visit(); ---------------------------------------------//057
--- } else if (_j < i or j < _i) { -------------------------//058
----- // do nothing ------------------------------------------//059
--- } else { -----------------------------------------------//05a
----- l->increase(_i, _j, _inc); ---------------------------//05b
----- r->increase(_i, _j, _inc); ---------------------------//05c
----- val = l->val + r->val;  }  } -------------------------//05d
- int query(int _i, int _j) { ------------------------------//05e
--- visit(); -----------------------------------------------//05f
--- if (_i <= i and j <= _j) { -----------------------------//060
----- return val; ------------------------------------------//061
--- } else if (_j < i || j < _i) { -------------------------//062
----- return 0; --------------------------------------------//063
--- } else { -----------------------------------------------//064
----- return l->query(_i, _j) + r->query(_i, _j);  }  }  };
```

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

```
struct node {  int l, r, lid, rid, val;  }; ----------------//066
struct segtree { -------------------------------------------//067
- node *nodes; ---------------------------------------------//068
- int n, node_cnt = 0; -------------------------------------//069
- segtree(int n, int capacity) { ---------------------------//06a
--- this->n = n; -------------------------------------------//06b
--- nodes = new node[capacity];  } ------------------------//06c
- int build (int *ar, int l, int r) { ----------------------//06d
--- if (l > r)  return -1; ---------------------------------//06e
--- int id = node_cnt++; -----------------------------------//06f
--- nodes[id].l = l; ---------------------------------------//070
--- nodes[id].r = r; ---------------------------------------//071
--- if (l == r) { ------------------------------------------//072
----- nodes[id].lid = -1; ----------------------------------//073
----- nodes[id].rid = -1; ----------------------------------//074
----- nodes[id].val = ar[l]; -------------------------------//075
--- } else { -----------------------------------------------//076
----- int m = (l + r) / 2; ---------------------------------//077
----- nodes[id].lid = build(ar, l, m); ---------------------//078
----- nodes[id].rid = build(ar, m+1, r); -------------------//079
----- nodes[id].val = nodes[nodes[id].lid].val + -----------//07a
-------------------- nodes[nodes[id].rid].val;  } ---------//07b
--- return id;  } ------------------------------------------//07c
- int update(int id, int idx, int delta) { -----------------//07d
--- if (id == -1) ------------------------------------------//07e
----- return -1; -------------------------------------------//07f
--- if (idx < nodes[id].l or nodes[id].r < idx) ------------//080
----- return id; -------------------------------------------//081
--- int nid = node_cnt++; ----------------------------------//082
--- nodes[nid].l = nodes[id].l; ----------------------------//083
--- nodes[nid].r = nodes[id].r; ----------------------------//084
--- nodes[nid].lid = update(nodes[id].lid, idx, delta); ---//085
--- nodes[nid].rid = update(nodes[id].rid, idx, delta); ---//086
--- nodes[nid].val = nodes[id].val + delta; ---------------//087
--- return nid;  } -----------------------------------------//088
- int query(int id, int l, int r) { ------------------------//089
--- if (r < nodes[id].l or nodes[id].r < l) ----------------//08a
----- return 0; --------------------------------------------//08b
--- if (l <= nodes[id].l and nodes[id].r <= r) ------------//08c
----- return nodes[id].val; --------------------------------//08d
--- return query(nodes[id].lid, l, r) + -------------------//08e
--------- query(nodes[id].rid, l, r);  }  }; --------------//08f
```

2.4. **Sparse Table.**

2.5. **Sqrt Decomposition.**

2.6. **Treap.**

2.6.1. *Explicit Treap.*

2.6.2. *Implicit Treap.*

```
struct cartree { -------------------------------------------//0ac
- typedef struct _Node { -----------------------------------//0ad
--- int node_val, subtree_val, delta, prio, size; ---------//0ae
--- _Node *l, *r; ------------------------------------------//0af
--- _Node(int val) : node_val(val), subtree_val(val), ----//0b0
------- delta(0), prio((rand()<<16)^rand()), size(1), ----//0b1
------- l(NULL), r(NULL) {} --------------------------------//0b2
--- ~_Node() { delete l; delete r; } -----------------------//0b3
- } *Node; -------------------------------------------------//0b4
- int get_subtree_val(Node v) { ----------------------------//0b5
--- return v ? v->subtree_val : 0;  } ----------------------//0b6
- int get_size(Node v) { return v ? v->size : 0; } --------//0b7
- void apply_delta(Node v, int delta) { --------------------//0b8
--- if (!v) return; ----------------------------------------//0b9
--- v->delta += delta; -------------------------------------//0ba
--- v->node_val += delta; ----------------------------------//0bb
--- v->subtree_val += delta * get_size(v);  } -------------//0bc
- void push_delta(Node v) { --------------------------------//0bd
--- if (!v) return; ----------------------------------------//0be
--- apply_delta(v->l, v->delta); ---------------------------//0bf
--- apply_delta(v->r, v->delta); ---------------------------//0c0
--- v->delta = 0;  } ---------------------------------------//0c1
- void update(Node v) { ------------------------------------//0c2
--- if (!v) return; ----------------------------------------//0c3
--- v->subtree_val = get_subtree_val(v->l) + v->node_val -//0c4
------------------- + get_subtree_val(v->r); --------------//0c5
--- v->size = get_size(v->l) + 1 + get_size(v->r); } -----//0c6
- Node merge(Node l, Node r) { -----------------------------//0c7
--- push_delta(l);    push_delta(r); -----------------------//0c8
--- if (!l || !r)   return l ? l : r; ----------------------//0c9
--- if (l->size <= r->size) { ------------------------------//0ca
----- l->r = merge(l->r, r); -------------------------------//0cb
----- update(l); -------------------------------------------//0cc
----- return l; --------------------------------------------//0cd
--- } else { -----------------------------------------------//0ce
----- r->l = merge(l, r->l); -------------------------------//0cf
----- update(r); -------------------------------------------//0d0
----- return r;  }  } --------------------------------------//0d1
- void split(Node v, int key, Node &l, Node &r) { ----------//0d2
--- push_delta(v); -----------------------------------------//0d3
--- l = r = NULL; ------------------------------------------//0d4
--- if (!v)     return; ------------------------------------//0d5
--- if (key <= get_size(v->l)) { ---------------------------//0d6
----- split(v->l, key, l, v->l); ---------------------------//0d7
----- r = v; -----------------------------------------------//0d8
--- } else { -----------------------------------------------//0d9
----- split(v->r, key - get_size(v->l) - 1, v->r, r); ----//0da
----- l = v;  } --------------------------------------------//0db
--- update(v);  } ------------------------------------------//0dc
- Node root; -----------------------------------------------//0dd
public: ----------------------------------------------------//0de
- cartree() : root(NULL) {} --------------------------------//0df
- ~cartree() { delete root; } ------------------------------//0e0
- int get(Node v, int key) { -------------------------------//0e1
--- push_delta(v); -----------------------------------------//0e2
```

```
--- if (key < get_size(v->l)) ----------------------------//0e3
----- return get(v->l, key); -----------------------------//0e4
--- else if (key > get_size(v->l)) -----------------------//0e5
----- return get(v->r, key - get_size(v->l) - 1); --------//0e6
--- return v->node_val;  } -------------------------------//0e7
- int get(int key) { return get(root, key); } ------------//0e8
- void insert(Node item, int key) { ----------------------//0e9
--- Node l, r; -------------------------------------------//0ea
--- split(root, key, l, r); ------------------------------//0eb
--- root = merge(merge(l, item), r);  } ------------------//0ec
- void insert(int key, int val) { ------------------------//0ed
--- insert(new _Node(val), key);  } ----------------------//0ee
- void erase(int key) { ----------------------------------//0ef
--- Node l, m, r; ----------------------------------------//0f0
--- split(root, key + 1, m, r); --------------------------//0f1
--- split(m, key, l, m); ---------------------------------//0f2
--- delete m; --------------------------------------------//0f3
--- root = merge(l, r);  } -------------------------------//0f4
- int query(int a, int b) { ------------------------------//0f5
--- Node l1, r1; -----------------------------------------//0f6
--- split(root, b+1, l1, r1); ----------------------------//0f7
--- Node l2, r2; -----------------------------------------//0f8
--- split(l1, a, l2, r2); --------------------------------//0f9
--- int res = get_subtree_val(r2); -----------------------//0fa
--- l1 = merge(l2, r2); ----------------------------------//0fb
--- root = merge(l1, r1); --------------------------------//0fc
--- return res;  } ---------------------------------------//0fd
- int update(int a, int b, int delta) { ------------------//0fe
--- Node l1, r1; -----------------------------------------//0ff
--- split(root, b+1, l1, r1); ----------------------------//100
--- Node l2, r2; -----------------------------------------//101
--- split(l1, a, l2, r2); --------------------------------//102
--- apply_delta(r2, delta); ------------------------------//103
--- l1 = merge(l2, r2); ----------------------------------//104
--- root = merge(l1, r1);  } ------------------------------//105
- int size() { return get_size(root); }  }; --------------//106
```

2.6.3. *Persistent Treap.*

2.7. Ordered Statistics Tree.

2.8. Union Find.

```
struct union_find { --------------------------------------//107
- vi p; union_find(int n) : p(n, -1) { } -----------------//108
- int find(int x) { return p[x] < 0 ? x : p[x] = find(p[x]); }
- bool unite(int x, int y) { -----------------------------//10a
--- int xp = find(x), yp = find(y); ----------------------//10b
--- if (xp == yp) return false; --------------------------//10c
--- if (p[xp] > p[yp]) swap(xp,yp); ----------------------//10d
--- p[xp] += p[yp], p[yp] = xp; --------------------------//10e
--- return true; } ---------------------------------------//10f
- int size(int x) { return -p[find(x)]; } }; -------------//110
```

3. Graphs

Using adjacency list:

```
struct graph { -------------------------------------------//1a7
- int n; -------------------------------------------------//1a8
- vii *adj; ----------------------------------------------//1a9
- int *dist; ---------------------------------------------//1aa
- graph(int n) { -----------------------------------------//1ab
--- this->n = n; -----------------------------------------//1ac
--- adj = new vii[n]; ------------------------------------//1ad
--- dist = new int[n]; } ---------------------------------//1ae
- void add_edge(int u, int v, int w) { -------------------//1af
--- adj[u].push_back({v, w}); ----------------------------//1b0
--- /*adj[v].push_back({u, w});*/  } }; ------------------//1b1
```

Using adjacency matrix:

```
struct graph { -------------------------------------------//1b2
- int n; -------------------------------------------------//1b3
- int **mat; ---------------------------------------------//1b4
- graph(int n) { -----------------------------------------//1b5
--- this->n = n; -----------------------------------------//1b6
--- mat = new int*[n]; -----------------------------------//1b7
--- for (int i = 0; i < n; ++i) { ------------------------//1b8
----- mat[i] = new int[n]; -------------------------------//1b9
----- for (int j = 0; j < n; ++j) ------------------------//1ba
------- mat[i][j] = INF; ---------------------------------//1bb
----- mat[i][i] = 0;  }  } -------------------------------//1bc
- void add_edge(int u, int v, int w) { -------------------//1bd
--- mat[u][v] = std::min(mat[u][v], w); ------------------//1be
- /*mat[v][u] = std::min(mat[v][u], w);*/  } }; ----------//1bf
```

Using edge list:

```
struct edge { --------------------------------------------//1c0
- int u, v, w; -------------------------------------------//1c1
- edge(int u, int v, int w) : u(u), v(v), w(w) {} --------//1c2
- const bool operator <(const edge &other) const { -------//1c3
--- return w < other.w;  }  }; ---------------------------//1c4
struct graph { -------------------------------------------//1c5
- int n; -------------------------------------------------//1c6
- std::vector<edge> edges; -------------------------------//1c7
- graph(int n) : n(n) {} ---------------------------------//1c8
- void add_edge(int u, int v, int w) { -------------------//1c9
--- edges.push_back(edge(u, v, w));  } }; ----------------//1ca
```

3.1. Single-Source Shortest Paths.

3.1.1. *Dijkstra.*

```
#include "graph_template_adjlist.cpp" --------------------//122
void dijkstra(int s, int n, int *dist, vii *adj) { -------//123
- for (int u = 0; u < n; ++u) ----------------------------//124
--- dist[u] = INF; ---------------------------------------//125
- dist[s] = 0; -------------------------------------------//126
- std::priority_queue<ii, vii, std::greater<ii> > pq; ----//127
- pq.push({0, s}); ---------------------------------------//128
- while (!pq.empty()) { ----------------------------------//129
--- int u = pq.top().second; -----------------------------//12a
--- int d = pq.top().first; ------------------------------//12b
--- pq.pop(); --------------------------------------------//12c
--- if (dist[u] < d) -------------------------------------//12d
----- continue; ------------------------------------------//12e
--- dist[u] = d; -----------------------------------------//12f
--- for (auto &e : adj[u]) { -----------------------------//130
----- int v = e.first; -----------------------------------//131
----- int w = e.second; ----------------------------------//132
----- if (dist[v] > dist[u] + w) { -----------------------//133
------- dist[v] = dist[u] + w; ---------------------------//134
------- pq.push({dist[v], v});  }  }  }  } ---------------//135
```

3.1.2. *Bellman-Ford.*

```
#include "graph_template_adjlist.cpp" --------------------//111
void bellman_ford(int s, int n, int *dist, vii *adj) { ---//112
- for (int u = 0; u < n; ++u) ----------------------------//113
--- dist[u] = INF; ---------------------------------------//114
- dist[s] = 0; -------------------------------------------//115
- for (int i = 0; i < n-1; ++i) --------------------------//116
--- for (int u = 0; u < n; ++u) --------------------------//117
----- for (auto &e : adj[u]) -----------------------------//118
------- if (dist[u] + e.second < dist[e.first]) ----------//119
--------- dist[e.first] = dist[u] + e.second;  } ---------//11a
// you can call this after running bellman_ford() --------//11b
bool has_neg_cycle(int n, int *dist, vii *adj) { ---------//11c
- for (int u = 0; u < n; ++u) ----------------------------//11d
--- for (auto &e : adj[u]) -------------------------------//11e
----- if (dist[e.first] > dist[u] + e.second) ------------//11f
------- return true; -------------------------------------//120
- return false;  } ---------------------------------------//121
```

3.2. All-Pairs Shortest Paths.

3.2.1. *Floyd-Washall.*

```
#include "graph_template_adjmat.cpp" ---------------------//1a0
void floyd_warshall(int n, int **mat) { ------------------//1a1
- for (int k = 0; k < n; ++k) ----------------------------//1a2
--- for (int i = 0; i < n; ++i) --------------------------//1a3
----- for (int j = 0; j < n; ++j) ------------------------//1a4
------- if (mat[i][k] + mat[k][j] < mat[i][j]) -----------//1a5
--------- mat[i][j] = mat[i][k] + mat[k][j];  } ----------//1a6
```

3.3. Strongly Connected Components.

3.3.1. *Kosaraju.*

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 max_flow { ------------------------------------------//173
- int n, s, t, *par, **c, **f; -----------------------------//174
- vi *adj; -------------------------------------------------//175
- max_flow(int n, int s, int t) : n(n), s(s), t(t) { -----//176
--- adj = new std::vector<int>[n]; ------------------------//177
--- par = new int[n]; -------------------------------------//178
--- c = new int*[n]; --------------------------------------//179
--- f = new int*[n]; --------------------------------------//17a
--- for (int i = 0; i < n; ++i) { -------------------------//17b
----- c[i] = new int[n]; ----------------------------------//17c
----- f[i] = new int[n]; ----------------------------------//17d
----- for (int j = 0; j < n; ++j) -------------------------//17e
------- c[i][j] = f[i][j] = 0;  }  } ----------------------//17f
- void add_edge(int u, int v, int w) { --------------------//180
--- adj[u].push_back(v); ----------------------------------//181
--- adj[v].push_back(u); ----------------------------------//182
--- c[u][v] += w;  } --------------------------------------//183
- int res(int i, int j) { return c[i][j] - f[i][j]; } ----//184
- bool bfs() { --------------------------------------------//185
--- std::queue<int> q; ------------------------------------//186
--- q.push(this->s); --------------------------------------//187
--- while (!q.empty()) { ----------------------------------//188
----- int u = q.front();  q.pop(); ------------------------//189
----- for (int v : adj[u]) { ------------------------------//18a
------- if (res(u, v) > 0 and par[v] == -1) { ------------//18b
--------- par[v] = u; ------------------------------------//18c
--------- if (v == this->t) ------------------------------//18d
----------- return true; ---------------------------------//18e
--------- q.push(v);  }  }  } ----------------------------//18f
--- return false;  } --------------------------------------//190
- bool aug_path() { ---------------------------------------//191
--- for (int u = 0; u < n; ++u) --------------------------//192
----- par[u] = -1; ---------------------------------------//193
--- par[s] = s; ------------------------------------------//194
--- return bfs();  } -------------------------------------//195
- int calc_max_flow() { ----------------------------------//196
--- int ans = 0; -----------------------------------------//197
--- while (aug_path()) { ---------------------------------//198
----- int flow = INF; ------------------------------------//199
----- for (int u = t; u != s; u = par[u]) ----------------//19a
------- flow = std::min(flow, res(par[u], u)); -----------//19b
----- for (int u = t; u != s; u = par[u]) ----------------//19c
------- f[par[u]][u] += flow, f[u][par[u]] -= flow; ------//19d
----- ans += flow;  } ------------------------------------//19e
--- return ans;  }  }; -----------------------------------//19f
```

3.10.2. *Dinic.*

```
struct max_flow { ------------------------------------------//136
- int n, s, t, *adj_ptr, *dist, *par, **c, **f; -----------//137
- vi *adj; -------------------------------------------------//138
- max_flow(int n, int s, int t) : n(n), s(s), t(t) { -----//139
--- adj = new std::vector<int>[n]; ------------------------//13a
--- adj_ptr = new int[n]; ---------------------------------//13b
--- dist = new int[n]; ------------------------------------//13c
--- par = new int[n]; -------------------------------------//13d
--- c = new int*[n]; --------------------------------------//13e
--- f = new int*[n]; --------------------------------------//13f
--- for (int i = 0; i < n; ++i) { -------------------------//140
----- c[i] = new int[n]; ----------------------------------//141
----- f[i] = new int[n]; ----------------------------------//142
----- for (int j = 0; j < n; ++j) -------------------------//143
------- c[i][j] = f[i][j] = 0;  }  } ----------------------//144
- void add_edge(int u, int v, int w) { --------------------//145
--- adj[u].push_back(v); ----------------------------------//146
--- adj[v].push_back(u); ----------------------------------//147
--- c[u][v] += w;  } --------------------------------------//148
- int res(int i, int j) { return c[i][j] - f[i][j]; } ----//149
- void reset(int *ar, int val) { --------------------------//14a
--- for (int i = 0; i < n; ++i) --------------------------//14b
----- ar[i] = val;  } ------------------------------------//14c
- bool make_level_graph() { ------------------------------//14d
--- reset(dist, -1); -------------------------------------//14e
--- std::queue<int> q; -----------------------------------//14f
--- q.push(s); -------------------------------------------//150
--- dist[s] = 0; -----------------------------------------//151
--- while (!q.empty()) { ---------------------------------//152
----- int u = q.front();  q.pop(); -----------------------//153
----- for (int v : adj[u]) { -----------------------------//154
------- if (res(u, v) > 0 and dist[v] == -1) { -----------//155
--------- dist[v] = dist[u] + 1; -------------------------//156
--------- q.push(v);  }  }  } ----------------------------//157
--- return dist[t] != -1;  } -----------------------------//158
- bool next(int u, int v) { return dist[v] == dist[u] + 1; }
- bool dfs(int u) { --------------------------------------//15a
--- if (u == t)   return true; ---------------------------//15b
--- for (int &i = adj_ptr[u]; i < adj[u].size(); ++i) { --//15c
----- int v = adj[u][i]; ---------------------------------//15d
----- if (next(u, v) and res(u, v) > 0 and dfs(v)) { -----//15e
------- par[v] = u; --------------------------------------//15f
------- return true;  }  } -------------------------------//160
--- dist[u] = -1; ----------------------------------------//161
--- return false;  } -------------------------------------//162
- bool aug_path() { --------------------------------------//163
--- reset(par, -1); --------------------------------------//164
--- par[s] = s; ------------------------------------------//165
--- return dfs(s);  } ------------------------------------//166
- int calc_max_flow() { ----------------------------------//167
--- int ans = 0; -----------------------------------------//168
--- while (make_level_graph()) { -------------------------//169
----- reset(adj_ptr, 0); ---------------------------------//16a
----- while (aug_path()) { -------------------------------//16b
------- int flow = INF; ----------------------------------//16c
------- for (int u = t; u != s; u = par[u]) --------------//16d
--------- flow = std::min(flow, res(par[u], u)); ---------//16e
------- for (int u = t; u != s; u = par[u]) --------------//16f
--------- f[par[u]][u] += flow, f[u][par[u]] -= flow; ----//170
------- ans += flow;  }  } -------------------------------//171
--- return ans;  }  }; -----------------------------------//172
```

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

3.11.1. *Gomory-Hu.*

3.12. **Heavy Light Decomposition.**

```
#include "segment_tree.cpp" --------------------------------//1cb
struct heavy_light_tree { ---------------------------------//1cc
- int n; --------------------------------------------------//1cd
- std::vector<int> *adj; ----------------------------------//1ce
- segtree *segment_tree; ----------------------------------//1cf
- int *par, *heavy, *dep, *path_root, *pos; ---------------//1d0
- heavy_light_tree(int n) { -------------------------------//1d1
--- this->n = n; ------------------------------------------//1d2
--- this->adj = new std::vector<int>[n]; ------------------//1d3
--- segment_tree = new segtree(0, n-1); -------------------//1d4
--- par = new int[n]; -------------------------------------//1d5
--- heavy = new int[n]; -----------------------------------//1d6
--- dep = new int[n]; -------------------------------------//1d7
--- path_root = new int[n]; -------------------------------//1d8
--- pos = new int[n];  } ----------------------------------//1d9
- void add_edge(int u, int v) { ---------------------------//1da
--- adj[u].push_back(v); ----------------------------------//1db
--- adj[v].push_back(u);  } -------------------------------//1dc
- void build(int root) { ----------------------------------//1dd
--- for (int u = 0; u < n; ++u) --------------------------//1de
----- heavy[u] = -1; -------------------------------------//1df
--- par[root] = root; ------------------------------------//1e0
--- dep[root] = 0; ---------------------------------------//1e1
--- dfs(root); -------------------------------------------//1e2
--- for (int u = 0, p = 0; u < n; ++u) { -----------------//1e3
----- if (par[u] == -1 or heavy[par[u]] != u) { ----------//1e4
------- for (int v = u; v != -1; v = heavy[v]) { ---------//1e5
--------- path_root[v] = u; ------------------------------//1e6
--------- pos[v] = p++;  }  }  }  } ----------------------//1e7
- int dfs(int u) { ---------------------------------------//1e8
--- int sz = 1; ------------------------------------------//1e9
--- int max_subtree_sz = 0; ------------------------------//1ea
--- for (int v : adj[u]) { -------------------------------//1eb
----- if (v != par[u]) { ---------------------------------//1ec
------- par[v] = u; --------------------------------------//1ed
------- dep[v] = dep[u] + 1; -----------------------------//1ee
------- int subtree_sz = dfs(v); -------------------------//1ef
------- if (max_subtree_sz < subtree_sz) { ---------------//1f0
--------- max_subtree_sz = subtree_sz; -------------------//1f1
--------- heavy[u] = v;  } -------------------------------//1f2
------- sz += subtree_sz;  }  } --------------------------//1f3
--- return sz;  } ----------------------------------------//1f4
- int query(int u) { -------------------------------------//1f5
--- return segment_tree->sum(pos[u], pos[u]);  } ---------//1f6
- void update(int u, int v, int c) { ---------------------//1f7
--- for (; path_root[u] != path_root[v]; -----------------//1f8
---------- v = par[path_root[v]]) { ----------------------//1f9
----- if (dep[path_root[u]] > dep[path_root[v]]) ---------//1fa
------- std::swap(u, v); ---------------------------------//1fb
----- segment_tree->increase(pos[path_root[v]], pos[v], c);  }
--- segment_tree->increase(pos[u], pos[v], c);  }  }; ----//1fd
```

3.13. **Centroid Decomposition.**

3.14. **Least Common Ancestor.**

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.*

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

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

$$pf(A) = \frac{1}{2^n n!} \sum_{\sigma \in S_{2n}} \operatorname{sgn}(\sigma) \prod_{i=1}^{n} a_{\sigma(2i-1),\sigma(2i)}$$

$$= \sum_{M \in \mathrm{PM}(n)} \operatorname{sgn}(M) \prod_{(i,j) \in M} a_{i,j}$$

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.