AdMU Progvar

Team Notebook

26/09/2019

Contents

1. Code Templates

```cpp
#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.*

```cpp
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.*

```cpp
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 {
- 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 {
- int n;
- int *vals;
- segtree(int *ar, int n) {
--- this->n = n;
--- vals = new int[2*n];
--- for (int i = 0; i < n; ++i)
----- vals[n+i] = ar[i];
--- for (int i = n-1; i > 0; --i)
----- vals[i] = vals[i<<1] + vals[i<<1|1];
- }
- void update(int i, int v) {
--- for (vals[i += n] = v; i > 1; i >>= 1)
----- vals[i>>1] = vals[i] + vals[i^1];
- }
- int query(int l, int r) {
--- r++;    // without this, the range is [l,r)
--- int res = 0;
--- for (l += n, r += n; l < r; l >>= 1, r >>= 1) {
----- if (l&1)  res += vals[l++];
----- if (r&1)  res += vals[--r];
--- }
--- return res;
- }
};
```

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

```
struct segtree {
- int i, j, val, temp_val = 0;
- 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 visit() {
--- if (temp_val) {
----- val += (j-i+1) * temp_val;
----- if (l) {
------- l->temp_val += temp_val;
------- r->temp_val += temp_val;
----- }
----- temp_val = 0;
--- }
- }
- void increase(int _i, int _j, int _inc) {
--- visit();
--- if (_i <= i && j <= _j) {
----- temp_val += _inc;
----- visit();
--- } else if (_j < i or j < _i) {
----- // do nothing
--- } else {
----- l->increase(_i, _j, _inc);
----- r->increase(_i, _j, _inc);
----- val = l->val + r->val;
--- }
- }
- int query(int _i, int _j) {
--- visit();
--- if (_i <= i and j <= _j) {
----- return val;
--- } else if (_j < i || j < _i) {
----- return 0;
--- } else {
----- 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;  };
struct segtree {
- node *nodes;
- int n, node_cnt = 0;
- segtree(int n, int capacity) {
--- this->n = n;
--- nodes = new node[capacity];
- }
- int build (int *ar, int l, int r) {
--- if (l > r)  return -1;
--- int id = node_cnt++;
--- nodes[id].l = l;
--- nodes[id].r = r;
--- if (l == r) {
----- nodes[id].lid = -1;
----- nodes[id].rid = -1;
----- nodes[id].val = ar[l];
--- } else {
----- int m = (l + r) / 2;
----- nodes[id].lid = build(ar, l, m);
----- nodes[id].rid = build(ar, m+1, r);
----- nodes[id].val = nodes[nodes[id].lid].val +
-------------------- nodes[nodes[id].rid].val;
--- }
--- return id;
- }
- int update(int id, int idx, int delta) {
--- if (id == -1)
----- return -1;
--- if (idx < nodes[id].l or nodes[id].r < idx)
----- return id;
--- int nid = node_cnt++;
--- nodes[nid].l = nodes[id].l;
--- nodes[nid].r = nodes[id].r;
--- nodes[nid].lid = update(nodes[id].lid, idx, delta);
--- nodes[nid].rid = update(nodes[id].rid, idx, delta);
--- nodes[nid].val = nodes[id].val + delta;
--- return nid;
- }
- int query(int id, int l, int r) {
--- if (r < nodes[id].l or nodes[id].r < l)
----- return 0;
--- if (l <= nodes[id].l and nodes[id].r <= r)
----- return nodes[id].val;
--- return query(nodes[id].lid, l, r) +
--------- query(nodes[id].rid, l, r);
-- }
 };
```

2.4. **Sparse Table.**

2.5. **Sqrt Decomposition.**

2.6. **Treap.**

2.6.1. *Explicit Treap.*

2.6.2. *Implicit Treap.*

```
struct cartree {
- typedef struct _Node {
--- int node_val, subtree_val, delta, prio, size;
--- _Node *l, *r;
--- _Node(int val) : node_val(val), subtree_val(val),
------- delta(0), prio((rand()<<16)^rand()), size(1),
------- l(NULL), r(NULL) {}
--- ~_Node() { delete l; delete r; }
- } *Node;
- int get_subtree_val(Node v) {
--- return v ? v->subtree_val : 0;  }
- int get_size(Node v) { return v ? v->size : 0; }
- void apply_delta(Node v, int delta) {
--- if (!v) return;
--- v->delta += delta;
--- v->node_val += delta;
--- v->subtree_val += delta * get_size(v);
- }
- void push_delta(Node v) {
--- if (!v) return;
--- apply_delta(v->l, v->delta);
--- apply_delta(v->r, v->delta);
--- v->delta = 0;
- }
- void update(Node v) {
--- if (!v) return;
--- v->subtree_val = get_subtree_val(v->l) + v->node_val
----------------- + get_subtree_val(v->r);
--- v->size = get_size(v->l) + 1 + get_size(v->r);
- }
- Node merge(Node l, Node r) {
--- push_delta(l);    push_delta(r);
--- if (!l || !r)   return l ? l : r;
--- if (l->size <= r->size) {
----- l->r = merge(l->r, r);
----- update(l);
----- return l;
--- } else {
----- r->l = merge(l, r->l);
----- update(r);
----- return r;
--- }
- }
- void split(Node v, int key, Node &l, Node &r) {
--- push_delta(v);
--- l = r = NULL;
--- if (!v)     return;
--- if (key <= get_size(v->l)) {
----- split(v->l, key, l, v->l);
----- r = v;
--- } else {
----- split(v->r, key - get_size(v->l) - 1, v->r, r);
----- l = v;
--- }
--- update(v);
- }
- Node root;
public:
- cartree() : root(NULL) {}
- ~cartree() { delete root; }
- int get(Node v, int key) {
--- push_delta(v);
--- if (key < get_size(v->l))
----- return get(v->l, key);
--- else if (key > get_size(v->l))
----- return get(v->r, key - get_size(v->l) - 1);
--- return v->node_val;  }
- int get(int key) { return get(root, key); }
- void insert(Node item, int key) {
--- Node l, r;
--- split(root, key, l, r);
--- root = merge(merge(l, item), r);
- }
- void insert(int key, int val) {
--- insert(new _Node(val), key);
- }
- void erase(int key) {
--- Node l, m, r;
--- split(root, key + 1, m, r);
--- split(m, key, l, m);
--- delete m;
--- root = merge(l, r);
- }
- int query(int a, int b) {
--- Node l1, r1;
--- split(root, b+1, l1, r1);
--- Node l2, r2;
--- split(l1, a, l2, r2);
--- int res = get_subtree_val(r2);
--- l1 = merge(l2, r2);
--- root = merge(l1, r1);
--- return res;
- }
- int update(int a, int b, int delta) {
--- Node l1, r1;
--- split(root, b+1, l1, r1);
--- Node l2, r2;
--- split(l1, a, l2, r2);
--- apply_delta(r2, delta);
--- l1 = merge(l2, r2);
--- root = merge(l1, r1);
- }
- int size() { return get_size(root); }  };
```

2.6.3. *Persistent Treap.*

2.7. **Ordered Statistics Tree.**

2.8. **Union Find.**

```
struct union_find {
- vi p; union_find(int n) : p(n, -1) { }
- int find(int x) { return p[x] < 0 ? x : p[x] = find(p[x]); }
- bool unite(int x, int y) {
--- int xp = find(x), yp = find(y);
--- if (xp == yp) return false;
--- if (p[xp] > p[yp]) swap(xp,yp);
--- p[xp] += p[yp], p[yp] = xp;
--- return true;
- }
- int size(int x) { return -p[find(x)]; }
};
```

3. Graphs

Using adjacency list:

```
struct graph {
- int n;
- vii *adj;
- int *dist;
- graph(int n) {
--- this->n = n;
--- adj = new vii[n];
--- dist = new int[n];
- }
- void add_edge(int u, int v, int w) {
--- adj[u].push_back({v, w});
--- /*adj[v].push_back({u, w});*/
- }
};
```

Using adjacency matrix:

```
struct graph {
- int n;
- int **mat;
- graph(int n) {
--- this->n = n;
--- mat = new int*[n];
--- for (int i = 0; i < n; ++i) {
----- mat[i] = new int[n];
----- for (int j = 0; j < n; ++j)
------- mat[i][j] = INF;
----- mat[i][i] = 0;
--- }
- }
- void add_edge(int u, int v, int w) {
--- mat[u][v] = std::min(mat[u][v], w);
- /*mat[v][u] = std::min(mat[v][u], w);*/
- }
};
```

Using edge list:

```
struct edge {
- int u, v, w;
- edge(int u, int v, int w) : u(u), v(v), w(w) {}
- const bool operator <(const edge &other) const {
--- return w < other.w;
- }
```

```
}; ------------------------------------------------------//1b4
struct graph { ------------------------------------------//1b5
- int n; -----------------------------------------------//1b6
- std::vector<edge> edges; -----------------------------//1b7
- graph(int n) : n(n) {} -------------------------------//1b8
- void add_edge(int u, int v, int w) { -----------------//1b9
--- edges.push_back(edge(u, v, w)); --------------------//1ba
- } ----------------------------------------------------//1bb
}; ------------------------------------------------------//1bc
```

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" --------------------//185
// insert inside graph; needs n and mat[][] -------------//186
void floyd_warshall() { --------------------------------//187
- for (int k = 0; k < n; ++k) --------------------------//188
--- for (int i = 0; i < n; ++i) ------------------------//189
----- for (int j = 0; j < n; ++j) ----------------------//18a
------- if (mat[i][k] + mat[k][j] < mat[i][j]) ---------//18b
--------- mat[i][j] = mat[i][k] + mat[k][j]; -----------//18c
} -------------------------------------------------------//18d
```

3.3. Strongly Connected Components.

3.3.1. *Kosaraju.*

```
struct kosaraju_graph { --------------------------------//206
- int n; -----------------------------------------------//207
- int *vis; --------------------------------------------//208
- vi **adj; --------------------------------------------//209
- std::vector<vi> sccs; --------------------------------//20a
- kosaraju_graph(int n) { ------------------------------//20b
--- this->n = n; ---------------------------------------//20c
--- vis = new int[n]; ----------------------------------//20d
--- adj = new vi*[2]; ----------------------------------//20e
--- for (int dir = 0; dir < 2; ++dir) ------------------//20f
----- adj[dir] = new vi[n]; ----------------------------//210
- } ----------------------------------------------------//211
- void add_edge(int u, int v) { ------------------------//212
--- adj[0][u].push_back(v); ----------------------------//213
--- adj[1][v].push_back(u); ----------------------------//214
- } ----------------------------------------------------//215
- void dfs(int u, int p, int dir, vi &topo) { ----------//216
--- vis[u] = 1; ----------------------------------------//217
--- for (int v : adj[dir][u]) --------------------------//218
----- if (!vis[v] && v != p) ---------------------------//219
------- dfs(v, u, dir, topo); --------------------------//21a
--- topo.push_back(u); ---------------------------------//21b
- } ----------------------------------------------------//21c
- void kosaraju() { ------------------------------------//21d
--- vi topo; -------------------------------------------//21e
--- for (int u = 0; u < n; ++u)  vis[u] = 0; -----------//21f
--- for (int u = 0; u < n; ++u) ------------------------//220
----- if (!vis[u]) -------------------------------------//221
------- dfs(u, -1, 0, topo); ---------------------------//222
--- for (int u = 0; u < n; ++u)  vis[u] = 0; -----------//223
--- for (int i = n-1; i >= 0; --i) { -------------------//224
----- if (!vis[topo[i]]) { -----------------------------//225
------- sccs.push_back({}); ----------------------------//226
------- dfs(topo[i], -1, 1, sccs.back()); --------------//227
----- } ------------------------------------------------//228
--- } --------------------------------------------------//229
- } ----------------------------------------------------//22a
}; ------------------------------------------------------//22b
```

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 { ----------------------------------//2ae
- int n, s, t, *par, **c, **f; -------------------------//2af
- vi *adj; ---------------------------------------------//2b0
- flow_network(int n, int s, int t) : n(n), s(s), t(t) { -//2b1
--- adj = new std::vector<int>[n]; ---------------------//2b2
--- par = new int[n]; ----------------------------------//2b3
--- c = new int*[n]; -----------------------------------//2b4
--- f = new int*[n]; -----------------------------------//2b5
--- for (int i = 0; i < n; ++i) { ----------------------//2b6
----- c[i] = new int[n]; -------------------------------//2b7
----- f[i] = new int[n]; -------------------------------//2b8
----- for (int j = 0; j < n; ++j) ----------------------//2b9
------- c[i][j] = f[i][j] = 0; -------------------------//2ba
--- } --------------------------------------------------//2bb
- } ----------------------------------------------------//2bc
- void add_edge(int u, int v, int w) { -----------------//2bd
--- adj[u].push_back(v); -------------------------------//2be
--- adj[v].push_back(u); -------------------------------//2bf
--- c[u][v] += w; --------------------------------------//2c0
- } ----------------------------------------------------//2c1
- int res(int i, int j) { return c[i][j] - f[i][j]; } ---//2c2
- bool bfs() { -----------------------------------------//2c3
--- std::queue<int> q; ---------------------------------//2c4
--- q.push(this->s); -----------------------------------//2c5
--- while (!q.empty()) { -------------------------------//2c6
----- int u = q.front();  q.pop(); ---------------------//2c7
----- for (int v : adj[u]) { ---------------------------//2c8
------- if (res(u, v) > 0 and par[v] == -1) { ----------//2c9
--------- par[v] = u; ----------------------------------//2ca
--------- if (v == this->t) ----------------------------//2cb
----------- return true; -------------------------------//2cc
--------- q.push(v); -----------------------------------//2cd
------- } ----------------------------------------------//2ce
----- } ------------------------------------------------//2cf
--- } --------------------------------------------------//2d0
```

```
--- return false; -----------------------------------------//2d1
- } -------------------------------------------------------//2d2
- bool aug_path() { ---------------------------------------//2d3
--- for (int u = 0; u < n; ++u) --------------------------//2d4
----- par[u] = -1; ----------------------------------------//2d5
--- par[s] = s; -------------------------------------------//2d6
--- return bfs(); -----------------------------------------//2d7
- } -------------------------------------------------------//2d8
- int calc_max_flow() { -----------------------------------//2d9
--- int ans = 0; ------------------------------------------//2da
--- while (aug_path()) { ----------------------------------//2db
----- int flow = INF; -------------------------------------//2dc
----- for (int u = t; u != s; u = par[u]) ----------------//2dd
------- flow = std::min(flow, res(par[u], u)); -----------//2de
----- for (int u = t; u != s; u = par[u]) ----------------//2df
------- f[par[u]][u] += flow, f[u][par[u]] -= flow; ------//2e0
----- ans += flow; ----------------------------------------//2e1
--- } -----------------------------------------------------//2e2
--- return ans; -------------------------------------------//2e3
- } -------------------------------------------------------//2e4
}; --------------------------------------------------------//2e5
```

3.10.2. *Dinic.*

```
struct flow_network { -------------------------------------//260
- int n, s, t, *adj_ptr, *dist, *par, **c, **f; ----------//261
- vi *adj; ------------------------------------------------//262
- flow_network(int n, int s, int t) : n(n), s(s), t(t) { -//263
--- adj = new std::vector<int>[n]; -----------------------//264
--- adj_ptr = new int[n]; --------------------------------//265
--- dist = new int[n]; -----------------------------------//266
--- par = new int[n]; ------------------------------------//267
--- c = new int*[n]; -------------------------------------//268
--- f = new int*[n]; -------------------------------------//269
--- for (int i = 0; i < n; ++i) { ------------------------//26a
----- c[i] = new int[n]; ---------------------------------//26b
----- f[i] = new int[n]; ---------------------------------//26c
----- for (int j = 0; j < n; ++j) ------------------------//26d
------- c[i][j] = f[i][j] = 0; ---------------------------//26e
--- } -----------------------------------------------------//26f
- } -------------------------------------------------------//270
- void add_edge(int u, int v, int w) { -------------------//271
--- adj[u].push_back(v); ---------------------------------//272
--- adj[v].push_back(u); ---------------------------------//273
--- c[u][v] += w; ----------------------------------------//274
- } -------------------------------------------------------//275
- int res(int i, int j) { return c[i][j] - f[i][j]; } ----//276
- void reset(int *ar, int val) { -------------------------//277
--- for (int i = 0; i < n; ++i) --------------------------//278
----- ar[i] = val; ---------------------------------------//279
- } -------------------------------------------------------//27a
- bool make_level_graph() { ------------------------------//27b
--- reset(dist, -1); -------------------------------------//27c
--- std::queue<int> q; -----------------------------------//27d
--- q.push(s); -------------------------------------------//27e
--- dist[s] = 0; -----------------------------------------//27f
--- while (!q.empty()) { ---------------------------------//280
----- int u = q.front();  q.pop(); -----------------------//281
----- for (int v : adj[u]) { -----------------------------//282
------- if (res(u, v) > 0 and dist[v] == -1) { -----------//283
--------- dist[v] = dist[u] + 1; -------------------------//284
--------- q.push(v); -------------------------------------//285
------- } -------------------------------------------------//286
----- } ---------------------------------------------------//287
--- } -----------------------------------------------------//288
--- return dist[t] != -1; --------------------------------//289
- } -------------------------------------------------------//28a
- bool next(int u, int v) { ------------------------------//28b
--- return dist[v] == dist[u] + 1; -----------------------//28c
- } -------------------------------------------------------//28d
- bool dfs(int u) { --------------------------------------//28e
--- if (u == t)   return true; ---------------------------//28f
--- for (int &i = adj_ptr[u]; i < adj[u].size(); ++i) { --//290
----- int v = adj[u][i]; ---------------------------------//291
----- if (next(u, v) and res(u, v) > 0 and dfs(v)) { -----//292
------- par[v] = u; --------------------------------------//293
------- return true; -------------------------------------//294
----- } ---------------------------------------------------//295
--- } -----------------------------------------------------//296
--- dist[u] = -1; ----------------------------------------//297
--- return false; ----------------------------------------//298
- } -------------------------------------------------------//299
- bool aug_path() { ---------------------------------------//29a
--- reset(par, -1); --------------------------------------//29b
--- par[s] = s; ------------------------------------------//29c
--- return dfs(s);  } ------------------------------------//29d
- int calc_max_flow() { -----------------------------------//29e
--- int ans = 0; ------------------------------------------//29f
--- while (make_level_graph()) { --------------------------//2a0
----- reset(adj_ptr, 0); ---------------------------------//2a1
----- while (aug_path()) { --------------------------------//2a2
------- int flow = INF; ----------------------------------//2a3
------- for (int u = t; u != s; u = par[u]) --------------//2a4
--------- flow = std::min(flow, res(par[u], u)); ---------//2a5
------- for (int u = t; u != s; u = par[u]) --------------//2a6
--------- f[par[u]][u] += flow, f[u][par[u]] -= flow; ----//2a7
------- ans += flow; -------------------------------------//2a8
----- } ---------------------------------------------------//2a9
--- } -----------------------------------------------------//2aa
--- return ans; -------------------------------------------//2ab
- } -------------------------------------------------------//2ac
}; --------------------------------------------------------//2ad
```

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

3.11.1. *Gomory-Hu.*

3.12. **Heavy Light Decomposition.**

```
#include "segment_tree.cpp" ------------------------------//1bd
struct heavy_light_tree { --------------------------------//1be
- int n; -------------------------------------------------//1bf
- std::vector<int> *adj; ---------------------------------//1c0
- segtree *segment_tree; ---------------------------------//1c1
- int *par, *heavy, *dep, *path_root, *pos; --------------//1c2
- heavy_light_tree(int n) { ------------------------------//1c3
--- this->n = n; -----------------------------------------//1c4
--- this->adj = new std::vector<int>[n]; -----------------//1c5
--- segment_tree = new segtree(0, n-1); ------------------//1c6
--- par = new int[n]; ------------------------------------//1c7
--- heavy = new int[n]; ----------------------------------//1c8
--- dep = new int[n]; ------------------------------------//1c9
--- path_root = new int[n]; ------------------------------//1ca
--- pos = new int[n]; ------------------------------------//1cb
- } -------------------------------------------------------//1cc
- void add_edge(int u, int v) { --------------------------//1cd
--- adj[u].push_back(v); ---------------------------------//1ce
--- adj[v].push_back(u); ---------------------------------//1cf
- } -------------------------------------------------------//1d0
- void build(int root) { ---------------------------------//1d1
--- for (int u = 0; u < n; ++u) --------------------------//1d2
----- heavy[u] = -1; -------------------------------------//1d3
--- par[root] = root; ------------------------------------//1d4
--- dep[root] = 0; ---------------------------------------//1d5
--- dfs(root); -------------------------------------------//1d6
--- for (int u = 0, p = 0; u < n; ++u) { -----------------//1d7
----- if (par[u] == -1 or heavy[par[u]] != u) { ----------//1d8
------- for (int v = u; v != -1; v = heavy[v]) { ---------//1d9
--------- path_root[v] = u; ------------------------------//1da
--------- pos[v] = p++; ----------------------------------//1db
------- } -------------------------------------------------//1dc
----- } ---------------------------------------------------//1dd
--- } -----------------------------------------------------//1de
- } -------------------------------------------------------//1df
- int dfs(int u) { ----------------------------------------//1e0
--- int sz = 1; -------------------------------------------//1e1
--- int max_subtree_sz = 0; ------------------------------//1e2
--- for (int v : adj[u]) { -------------------------------//1e3
----- if (v != par[u]) { ----------------------------------//1e4
------- par[v] = u; --------------------------------------//1e5
------- dep[v] = dep[u] + 1; -----------------------------//1e6
------- int subtree_sz = dfs(v); -------------------------//1e7
------- if (max_subtree_sz < subtree_sz) { ---------------//1e8
--------- max_subtree_sz = subtree_sz; -------------------//1e9
--------- heavy[u] = v; ----------------------------------//1ea
------- } -------------------------------------------------//1eb
------- sz += subtree_sz; --------------------------------//1ec
----- } ---------------------------------------------------//1ed
--- } -----------------------------------------------------//1ee
--- return sz; --------------------------------------------//1ef
- } -------------------------------------------------------//1f0
- int query(int u, int v) { ------------------------------//1f1
--- int res = 0; ------------------------------------------//1f2
--- while (path_root[u] != path_root[v]) { ---------------//1f3
----- if (dep[path_root[u]] > dep[path_root[v]]) ---------//1f4
------- std::swap(u, v); ---------------------------------//1f5
----- res += segment_tree->sum(pos[path_root[v]], pos[v]);//1f6
----- v = par[path_root[v]]; -----------------------------//1f7
--- } -----------------------------------------------------//1f8
--- res += segment_tree->sum(pos[u], pos[v]); ------------//1f9
--- return res; -------------------------------------------//1fa
```

```
- } ---------------------------------------------------//1fb
- void update(int u, int v, int c) { -------------------//1fc
--- for (; path_root[u] != path_root[v]; ----------------//1fd
---------- v = par[path_root[v]]) { ---------------------//1fe
----- if (dep[path_root[u]] > dep[path_root[v]]) ---------//1ff
------- std::swap(u, v); ---------------------------------//200
----- segment_tree->increase(pos[path_root[v]], pos[v], c);
--- } ---------------------------------------------------//202
--- segment_tree->increase(pos[u], pos[v], c); ----------//203
- } -----------------------------------------------------//204
}; ------------------------------------------------------//205
```

3.13. **Centroid Decomposition.**

3.14. **Least Common Ancestor.**

3.14.1. *Binary Lifting.*

```
struct graph { ------------------------------------------//22c
- int n; ------------------------------------------------//22d
- int logn; ---------------------------------------------//22e
- std::vector<int> *adj; --------------------------------//22f
- int *dep; ---------------------------------------------//230
- int **par; --------------------------------------------//231
- graph(int n, int logn=20) { ---------------------------//232
--- this->n = n; ----------------------------------------//233
--- this->logn = logn; ----------------------------------//234
--- adj = new std::vector<int>[n]; ----------------------//235
--- dep = new int[n]; -----------------------------------//236
--- par = new int*[n]; ----------------------------------//237
--- for (int i = 0; i < n; ++i) -------------------------//238
----- par[i] = new int[logn]; ---------------------------//239
- } -----------------------------------------------------//23a
- void dfs(int u, int p, int d) { -----------------------//23b
--- dep[u] = d; -----------------------------------------//23c
--- par[u][0] = p; --------------------------------------//23d
--- for (int v : adj[u]) --------------------------------//23e
----- if (v != p) ---------------------------------------//23f
------- dfs(v, u, d+1); ---------------------------------//240
- } -----------------------------------------------------//241
- int ascend(int u, int k) { ----------------------------//242
--- for (int i = 0; i < logn; ++i) ----------------------//243
----- if (k & (1 << i)) ---------------------------------//244
------- u = par[u][i]; ----------------------------------//245
--- return u; -------------------------------------------//246
- } -----------------------------------------------------//247
- int lca(int u, int v) { -------------------------------//248
--- if (dep[u] > dep[v])  u = ascend(u, dep[u] - dep[v]); //249
--- if (dep[v] > dep[u])  v = ascend(v, dep[v] - dep[u]); //24a
--- if (u == v)           return u; ---------------------//24b
--- for (int k = logn-1; k >= 0; --k) { -----------------//24c
----- if (par[u][k] != par[v][k]) { ---------------------//24d
------- u = par[u][k]; ----------------------------------//24e
------- v = par[v][k]; ----------------------------------//24f
----- } -------------------------------------------------//250
--- } ---------------------------------------------------//251
--- return par[u][0]; -----------------------------------//252
- } -----------------------------------------------------//253
- bool is_anc(int u, int v) { ---------------------------//254
--- if (dep[u] < dep[v]) --------------------------------//255
----- std::swap(u, v); ----------------------------------//256
--- return ascend(u, dep[u] - dep[v]) == v; -------------//257
- } -----------------------------------------------------//258
- void prep_lca(int root=0) { ---------------------------//259
--- dfs(root, root, 0); ---------------------------------//25a
--- for (int k = 1; k < logn; ++k) ----------------------//25b
----- for (int u = 0; u < n; ++u) -----------------------//25c
------- par[u][k] = par[par[u][k-1]][k-1]; --------------//25d
- } -----------------------------------------------------//25e
}; ------------------------------------------------------//25f
```

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; --------------------------//2e6
struct hasher { -----------------------------------------//2e7
- int n; ------------------------------------------------//2e8
- std::vector<ll> *p_pow; -------------------------------//2e9
- std::vector<ll> *h_ans; -------------------------------//2ea
- hash(vi &s, vi primes) { ------------------------------//2eb
--- n = primes.size(); ----------------------------------//2ec
--- p_pow = new std::vector<ll>[n]; ---------------------//2ed
--- h_ans = new std::vector<ll>[n]; ---------------------//2ee
--- for (int i = 0; i < n; ++i) { -----------------------//2ef
----- p_pow[i] = std::vector<ll>(MAXN); -----------------//2f0
----- p_pow[i][0] = 1; ----------------------------------//2f1
----- for (int j = 0; j+1 < MAXN; ++j) ------------------//2f2
------- p_pow[i][j+1] = (p_pow[i][j] * primes[i]) % MOD; -//2f3
----- h_ans[i] = std::vector<ll>(MAXN); -----------------//2f4
----- h_ans[i][0] = 0; ----------------------------------//2f5
----- for (int j = 0; j < s.size(); ++j) ----------------//2f6
------- h_ans[i][j+1] = (h_ans[i][j] + ------------------//2f7
----------------------- s[j] * p_pow[i][j]) % MOD; ------//2f8
--- } ---------------------------------------------------//2f9
- } -----------------------------------------------------//2fa
}; ------------------------------------------------------//2fb
```

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. $g(a_1, a_2) = a_1a_2 - a_1 - a_2$, $N(a_1, a_2) = (a_1-1)(a_2-1)/2$. $g(d \cdot a_1, d \cdot a_2, a_3) = d \cdot g(a_1, a_2, a_3) + a_3(d-1)$. 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.