

1 Graphs

1.1 Graph Traversal

1.1.1 Breadth First Search

```

1 void bfs(graph &g, int start) {
2     int n = g.size();
3     vi visited(n, 1);
4     queue<int> q;
5
6     q.emplace(start);
7     visited[start] = 0;
8     while (not q.empty()) {
9         int u = q.front();
10        q.pop();
11
12        for (int v : g[u]) {
13            if (visited[v]) {
14                q.emplace(v);
15                visited[v] = 0;
16            }
17        }
18    }
19 }
```

1.1.2 Recursive Depth First Search

```

1 // Recursive (create visited filled with 1s)
2 void dfs_r(graph &g, vi &visited, int u) {
3     cout << u << '\n';
4     visited[u] = 0;
5
6     for (int v : g[u])
7         if (visited[v])
8             dfs_r(g, visited, v);
9 }
```

1.1.3 Iterative Depth First Search

```

1 // Iterative
2 void dfs_i(graph &g, int start) {
3     int n = g.size();
4     vi visited(n, 1);
5     stack<int> s;
6
7     s.emplace(start);
8     visited[start] = 0;
9     while (not s.empty()) {
10        int u = s.top();
11        s.pop();
12
13        for (int v : g[u]) {
14            if (visited[v]) {
```

```

15        s.emplace(v);
16        visited[v] = 0;
17    }
18 }
19 }
20 }
```

1.2 Shortest Path Algorithms

1.2.1 Dijkstra

All edges have non-negative values

```

1 // g has vectors of pairs of the form (w, index)
2 int dijkstra(wgraph g, int start, int end) {
3     int n = g.size();
4     vi cost(n, 1e9); // ~INT_MAX/2
5     priority_queue<ii, greater<ii>> q;
6
7     q.emplace(0, start);
8     cost[start] = 0;
9     while (not q.empty()) {
10        int u = q.top().second, w = q.top().first;
11        q.pop();
12
13        // we skip all nodes in the q that we have discovered before at a
14        // lower cost
15        if (cost[u] < w)
16            continue;
17
18        for (auto v : g[u]) {
19            if (cost[v.second] > v.first + w) {
20                cost[v.second] = v.first + w;
21                q.emplace(cost[v.second], v.second);
22            }
23        }
24        return cost[end];
25    }
```

1.2.2 Bellman Ford

Edges can be negative, and it detects negative cycles

```

1 bool bellman_ford(wgraph &g, int start) {
2     int n = g.size();
3     vector<int> dist(n, 1e9); // ~INT_MAX/2
4     dist[start] = 0;
5     rep(i, n - 1) rep(u, n) for (ii p : g[u]) {
6         int v = p.first, w = p.second;
7         dist[v] = min(dist[v], dist[u] + w);
8     }
9
10    bool hayCicloNegativo = false;
11    rep(u, n) for (ii p : g[u]) {
```

```

12     int v = p.first, w = p.second;
13     if (dist[v] > dist[u] + w)
14         hayCicloNegativo = true;
15 }
16
17 return hayCicloNegativo;
18 }

```

1.2.3 Floyd Warshall

Shortest path from every node to every other node

```

1  /*
2  Floyd Warshall implemenation, note that g is using an adjacency matrix
3  and not
4  an adjacency list
5  */
6  static const int INF = 1e9;
7  graph floydWarshall(const graph g) {
8      int n = g.size();
9      graph dist(n, vi(n, -1));
10
11      rep(i, n) rep(j, n) dist[i][j] = g[i][j];
12
13      rep(k, n) rep(i, n) rep(j, n) if (dist[i][k] + dist[k][j] < dist[i][j]
14          &&
15          dist[i][k] != INF && dist[k][j] !=
16          INF)
17          dist[i][j] = dist[i][k] + dist[k][j];
18
19      return dist;
20 }

```

1.3 Minimum Spanning Tree (MST)

1.3.1 Kruskal

```

1 struct edge {
2     int u, v;
3     ll w;
4     edge(int u, int v, ll w) : u(u), v(v), w(w) {}
5
6     bool operator<(const edge &o) const { return w < o.w; }
7 };
8
9 class Kruskal {
10 private:
11     ll sum;
12     vi p, rank;
13
14 public:
15     // Amount of Nodes n, and unordered vector of Edges E
16     Kruskal(int n, vector<edge> E) {

```

```

17     sum = 0;
18     p.resize(n);
19     rank.assign(n, 0);
20     rep(i, n) p[i] = i;
21     sort(E.begin(), E.end());
22     for (auto &e : E)
23         UnionSet(e.u, e.v, e.w);
24 }
25 int findSet(int i) { return (p[i] == i) ? i : (p[i] = findSet(p[i])); }
26
27 bool isSameSet(int i, int j) { return findSet(i) == findSet(j); }
28 void UnionSet(int i, int j, ll w) {
29     if (not isSameSet(i, j)) {
30         int x = findSet(i), y = findSet(j);
31         if (rank[x] > rank[y])
32             p[y] = x;
33         else
34             p[x] = y;
35
36         if (rank[x] == rank[y])
37             rank[y]++;
38
39         sum += w;
40     }
41 }
42 ll mst_val() { return sum; }
43 };

```

1.4 Lowest Common Ancestor (LCA)

Supports multiple trees

```

1 class LcaForest {
2     int n;
3     vi parent;
4     vi level;
5     vi root;
6     graph P;
7
8 public:
9     LcaForest(int n) {
10         this->n = n;
11         parent.assign(n, -1);
12         level.assign(n, -1);
13         P.assign(n, vi(lg(n) + 1, -1));
14         root.assign(n, -1);
15     }
16
17     void addLeaf(int index, int par) {
18         parent[index] = par;
19         level[index] = level[par] + 1;
20         P[index][0] = par;
21         root[index] = root[par];
22         for (int j = 1; (1 << j) < n; ++j) {
23             if (P[index][j - 1] != -1)
24                 P[index][j] = P[P[index][j - 1]][j - 1];
25         }
26     }
27 }

```

```

25 }
26 void addRoot(int index) {
27     parent[index] = index;
28     level[index] = 0;
29     root[index] = index;
30 }
31 int lca(int u, int v) {
32     if (root[u] != root[v] || root[u] == -1)
33         return -1;
34     if (level[u] < level[v])
35         swap(u, v);
36     int dist = level[u] - level[v];
37     while (dist != 0) {
38         int raise = lg(dist);
39         u = P[u][raise];
40         dist -= (1 << raise);
41     }
42     if (u == v)
43         return u;
44     for (int j = lg(n); j >= 0; --j) {
45         if (P[u][j] != -1 && P[u][j] != P[v][j]) {
46             u = P[u][j];
47             v = P[v][j];
48         }
49     }
50     return parent[u];
51 }
52 };

```

1.5 Max Flow

```

1 class Dinic {
2     struct edge {
3         int to, rev;
4         ll f, cap;
5     };
6
7     vector<vector<edge>> g;
8     vector<ll> dist;
9     vector<int> q, work;
10    int n, sink;
11
12    bool bfs(int start, int finish) {
13        dist.assign(n, -1);
14        dist[start] = 0;
15        int head = 0, tail = 0;
16        q[tail++] = start;
17        while (head < tail) {
18            int u = q[head++];
19            for (const edge &e : g[u]) {
20                int v = e.to;
21                if (dist[v] == -1 and e.f < e.cap) {
22                    dist[v] = dist[u] + 1;
23                    q[tail++] = v;

```

```

24     }
25 }
26 }
27 return dist[finish] != -1;
28 }
29
30 ll dfs(int u, ll f) {
31     if (u == sink)
32         return f;
33     for (int &i = work[u]; i < (int)g[u].size(); ++i) {
34         edge &e = g[u][i];
35         int v = e.to;
36         if (e.cap <= e.f or dist[v] != dist[u] + 1)
37             continue;
38         ll df = dfs(v, min(f, e.cap - e.f));
39         if (df > 0) {
40             e.f += df;
41             g[v][e.rev].f -= df;
42             return df;
43         }
44     }
45     return 0;
46 }
47
48 public:
49     Dinic(int n) {
50         this->n = n;
51         g.resize(n);
52         dist.resize(n);
53         q.resize(n);
54     }
55
56     void add_edge(int u, int v, ll cap) {
57         edge a = {v, (int)g[v].size(), 0, cap};
58         edge b = {u, (int)g[u].size(), 0,
59                 0}; // Poner cap en vez de 0 si la arista es bidireccional
60         g[u].pb(a);
61         g[v].pb(b);
62     }
63
64     ll max_flow(int source, int dest) {
65         sink = dest;
66         ll ans = 0;
67         while (bfs(source, dest)) {
68             work.assign(n, 0);
69             while (ll delta = dfs(source, LLONG_MAX))
70                 ans += delta;
71         }
72         return ans;
73     }
74 };

```

1.6 Others

1.6.1 Diameter of a tree

```

1 graph Tree;
2 vi dist;
3
4 // Finds a diameter node
5 int bfs1() {
6     int n = Tree.size();
7     queue<int> q;
8
9     q.emplace(0);
10    dist[0] = 0;
11    int u;
12    while (not q.empty()) {
13        u = q.front();
14        q.pop();
15
16        for (int v : Tree[u]) {
17            if (dist[v] == -1) {
18                q.emplace(v);
19                dist[v] = dist[u] + 1;
20            }
21        }
22    }
23    return u;
24 }
25
26 // Fills the distances from one diameter node and finds another diameter
   node
27 int bfs2() {
28     int n = Tree.size();
29     vi visited(n, 1);
30     queue<int> q;
31     int start = bfs1();
32     q.emplace(start);
33     visited[start] = 0;
34     int u;
35     while (not q.empty()) {
36         u = q.front();
37         q.pop();
38
39         for (int v : Tree[u]) {
40             if (visited[v]) {
41                 q.emplace(v);
42                 visited[v] = 0;
43                 dist[v] = max(dist[v], dist[u] + 1);
44             }
45         }
46     }
47     return u;
48 }
49
50 // Finds the diameter
51 int bfs3() {
52     int n = Tree.size();
53     vi visited(n, 1);
54     queue<int> q;
55     int start = bfs2();
56     q.emplace(start);
57     visited[start] = 0;
58     int u;
59     while (not q.empty()) {
60         u = q.front();
61         q.pop();
62
63         for (int v : Tree[u]) {
64             if (visited[v]) {
65                 q.emplace(v);
66                 visited[v] = 0;
67                 dist[v] = max(dist[v], dist[u] + 1);
68             }
69         }
70     }
71     return dist[u];
72 }

```