1 Graphs

1.1 Graph Traversal

1.1.1 Breadth First Search

```
#include "../../headers/headers.h"

void bfs(graph &g, int start)
{
    int n = g.size();
    vi visited(n, 1);
    queue<int> q;

    q.emplace(start);
    visited[start] = 0;
    while (not q.empty())
    {
        int u = q.front();
        q.pop();

        for (int v : g[u])
        {
            if (visited[v])
            {
                q.emplace(v);
                visited[v] = 0;
            }
        }
    }
}
```

1.1.2 Recursive Depth First Search

```
#include "../../headers/headers.h"
//Recursive (create visited filled with 1s)
void dfs_r(graph &g, vi &visited, int u)
{
    cout << u << '\n';
    visited[u] = 0;

    for (int v : g[u])
        if (visited[v])
            dfs_r(g, visited, v);
}
```

1.1.3 Iterative Depth First Search

```
#include "../../headers/headers.h"
//Iterative
void dfs_i(graph &g, int start)
{
    int n = g.size();
    vi visited(n, 1);
    stack<int> s;

    s.emplace(start);
    visited[start] = 0;

    while (not s.empty())
    {
        int u = s.top();
        s.pop();

        cout << u << '\n';

        for (int v : g[u])
            if (visited[v])
            {
                s.emplace(v);
                visited[v] = 0;
            }
    }
}
```

1.2 Shortest Path Algorithms

1.2.1 Dijsktra

All edges have non-negative values

```
#include "../../headers/headers.h"
//g has vectors of pairs of the form (w, index)
int dijsktra(wgraph g, int start, int end)
{
    int n = g.size();
    vi cost(n, 1e9); //~INT_MAX/2
    priority_queue<ii, greater<ii>> q;

    q.emplace(0, start);
    cost[start] = 0;
    while (not q.empty())
    {
        int u = q.top().second, w = q.top().first;
        q.pop();

        // we skip all nodes in the q that we have discovered before at a lower cost
        if (cost[u] < w) continue;

        for (auto v : g[u])
        {
            if (cost[v.second] > v.first + w)
            {
                cost[v.second] = v.first + w;
                q.emplace(cost[v.second], v.second);
            }
        }
    }

```

```
    return cost[end];
}
```

1.2.2 Bellman Ford

Edges can be negative, and it detects negative cycles

```
#include "../../headers/headers.h"
bool bellman_ford(wgraph &g, int start)
{
    int n = g.size();
    vector<int> dist(n, 1e9); //~INT_MAX/2
    dist[start] = 0;
    rep(i, n - 1) rep(u, n) for (ii p : g[u])
    {
        int v = p.first, w = p.second;
        dist[v] = min(dist[v], dist[u] + w);
    }

    bool hayCicloNegativo = false;
    rep(u, n) for (ii p : g[u])
    {
        int v = p.first, w = p.second;
        if (dist[v] > dist[u] + w)
            hayCicloNegativo = true;
    }

    return hayCicloNegativo;
}
```

1.2.3 Floyd Warshall

Shortest path from every node to every other node

```
#include "../../headers/headers.h"

/*
Floyd Warshall implemenation, note that g is using an adjacency matrix
     and not an
adjacency list
*/
graph floydWarshall (const graph g)
{
    int n = g.size();
    graph dist(n, vi(n, -1));

    rep(i, n)
        rep(j, n)
            dist[i][j] = g[i][j];

    rep(k, n)
        rep(i, n)
            rep(j, n)
                if (dist[i][k] + dist[k][j] < dist[i][j] &&
                    dist[i][k] != INF                    &&
                    dist[k][j] != INF)
                    dist[i][j] = dist[i][k] + dist[k][j];

    return dist;
}
```

1.3 Minimum Spanning Tree (MST)

1.3.1 Kruskal

```
#include "../../headers/headers.h"
struct edge
{
    int u, v;
    ll w;
    edge(int u, int v, ll w) : u(u), v(v), w(w) {}

    bool operator<(const edge &o) const
    {
        return w < o.w;
    }
};

class Kruskal
{
  private:
    ll sum;
    vi p, rank;

  public:
  //Amount of Nodes n, and unordered vector of Edges E
    Kruskal(int n, vector<edge> E)
    {
        sum = 0;
        p.resize(n);
        rank.assign(n, 0);
        rep(i, n) p[i] = i;
        sort(E.begin(), E.end());
        for (auto &e : E)
            UnionSet(e.u, e.v, e.w);
    }
    int findSet(int i)
    {
        return (p[i] == i) ? i : (p[i] = findSet(p[i]));
    }
    bool isSameSet(int i, int j)
    {
        return findSet(i) == findSet(j);
    }
    void UnionSet(int i, int j, ll w)
    {
        if (not isSameSet(i, j))
        {
```

```
            int x = findSet(i), y = findSet(j);
            if (rank[x] > rank[y])
                p[y] = x;
            else
                p[x] = y;

            if (rank[x] == rank[y])
                rank[y]++;

            sum += w;
        }
    }
    ll mst_val()
    {
        return sum;
    }
};
```

1.4 Lowest Common Ancestor (LCA)

Supports multiple trees

```
#include "../../headers/headers.h"
class LcaForest
{
    int n;
    vi parent;
    vi level;
    vi root;
    graph P;

public:
    LcaForest(int n)
    {
        this->n = n;
        parent.assign(n, -1);
        level.assign(n, -1);
        P.assign(n, vi(lg(n) + 1, -1));
        root.assign(n, -1);
    }
    void addLeaf(int index, int par)
    {
        parent[index] = par;
        level[index] = level[par] + 1;
        P[index][0] = par;
        root[index] = root[par];
        for (int j = 1; (1 << j) < n; ++j)
        {
            if (P[index][j - 1] != -1)
                P[index][j] = P[P[index][j - 1]][j - 1];
        }
    }
    void addRoot(int index)
    {
        parent[index] = index;
        level[index] = 0;
        root[index] = index;
    }
    int lca(int u, int v)
    {
        if (root[u] != root[v] || root[u] == -1)
            return -1;
        if (level[u] < level[v])
            swap(u, v);
        int dist = level[u] - level[v];
        while (dist != 0)
        {
            int raise = lg(dist);
            u = P[u][raise];
            dist -= (1 << raise);
        }
        if (u == v)
            return u;
        for (int j = lg(n); j >= 0; --j)
        {
            if (P[u][j] != -1 && P[u][j] != P[v][j])
            {
                u = P[u][j];
                v = P[v][j];
            }
        }
        return parent[u];
    }
};
```

1.5 Max Flow

```
#include "../../headers/headers.h"
class Dinic
{
    struct edge
    {
        int to, rev;
        ll f, cap;
    };

    vector<vector<edge>> g;
    vector<ll> dist;
    vector<int> q, work;
    int n, sink;

    bool bfs(int start, int finish)
    {
        dist.assign(n, -1);
        dist[start] = 0;
        int head = 0, tail = 0;
        q[tail++] = start;
        while (head < tail)
        {
            int u = q[head++];
```

```cpp
            for (const edge &e : g[u])
            {
                int v = e.to;
                if (dist[v] == -1 and e.f < e.cap)
                {
                    dist[v] = dist[u] + 1;
                    q[tail++] = v;
                }
            }
        }
        return dist[finish] != -1;
    }

    ll dfs(int u, ll f)
    {
        if (u == sink)
            return f;
        for (int &i = work[u]; i < (int)g[u].size(); ++i)
        {
            edge &e = g[u][i];
            int v = e.to;
            if (e.cap <= e.f or dist[v] != dist[u] + 1)
                continue;
            ll df = dfs(v, min(f, e.cap - e.f));
            if (df > 0)
            {
                e.f += df;
                g[v][e.rev].f -= df;
                return df;
            }
        }
        return 0;
    }

  public:
    Dinic(int n)
    {
        this->n = n;
        g.resize(n);
        dist.resize(n);
        q.resize(n);
    }

    void add_edge(int u, int v, ll cap)
    {
        edge a = {v, (int)g[v].size(), 0, cap};
        edge b = {u, (int)g[u].size(), 0, 0}; //Poner cap en vez de 0 si la arista es bidireccional
        g[u].pb(a);
        g[v].pb(b);
    }

    ll max_flow(int source, int dest)
    {
        sink = dest;
        ll ans = 0;
        while (bfs(source, dest))
        {
            work.assign(n, 0);
            while (ll delta = dfs(source, LLONG_MAX))
                ans += delta;
        }
        return ans;
    }
};
```

1.6 Others

1.6.1 Diameter of a tree

```cpp
#include "../../headers/headers.h"

graph Tree;
vi dist;

// Finds a diameter node
int bfs1()
{
    int n = Tree.size();
    queue<int> q;

    q.emplace(0);
    dist[0] = 0;
    int u;
    while (not q.empty())
    {
        u = q.front();
        q.pop();

        for (int v : Tree[u])
        {
            if (dist[v] == -1)
            {
                q.emplace(v);
                dist[v] = dist[u] + 1;
            }
        }
    }
    return u;
}

// Fills the distances from one diameter node and finds another diameter node
int bfs2()
{
    int n = Tree.size();
    vi visited(n, 1);
    queue<int> q;
    int start = bfs1();
    q.emplace(start);
```

```
    visited[start] = 0;
    int u;
    while (not q.empty())
    {
        u = q.front();
        q.pop();

        for (int v : Tree[u])
        {
            if (visited[v])
            {
                q.emplace(v);
                visited[v] = 0;
                dist[v] = max(dist[v], dist[u] + 1);
            }
        }
    }
    return u;
}

// Finds the diameter
int bfs3()
{
    int n = Tree.size();
    vi visited(n, 1);
    queue<int> q;
    int start = bfs2();
    q.emplace(start);
    visited[start] = 0;
    int u;
    while (not q.empty())
    {
        u = q.front();
        q.pop();

        for (int v : Tree[u])
        {
            if (visited[v])
            {
                q.emplace(v);
                visited[v] = 0;
                dist[v] = max(dist[v], dist[u] + 1);
            }
        }
    }
    return dist[u];
}
```