```cpp
#include <bits/stdc++.h>

using namespace std;

#define BOOST cin.tie(0), cout.tie(0), ios_base::sync_with_stdio(false);
#define ll long long

#define vi vector<int>
#define vll vector<ll>
#define vd vector<double>
#define vb vector<bool>

#define vvi vector<vector<int>>
#define vvll vector<vector<ll>>
#define pii pair<int, int>

#define pll pair<ll, ll>
#define vpii vector<pii>
#define vpll vector<pll>

#define vs vector<string>
const int INF = 1e9;
const double EPS = 1e-6;
#define endl "\n"

#define format(n) fixed << setprecision(rows)
// DFS ***********************************************************************************
vvi adj_list;
vi visited;
int nodes, edges;

void dfsConnected(int u) {
    visited[u] = 1;
    for (auto neig: adj_list[u]) {
        if (!visited[neig]) {
            dfsConnected(neig);
        }
    }
}

/// count the number of connected components in undirected graph
void countConnected() {
    int components = 0;
    for (int i = 0; i < nodes; ++i) {
        if (!visited[i]) {
            dfsConnected(i);
            components++;
        }
    }

    cout << components << endl;
}

/// **************************************************************************************************************
*
bool dfsUndirected(int u, int parent) {
    visited[u] = true;
    bool cyclic = false;
    for (auto neig: adj_list[u]) {
        if (!visited[neig]) {
            cyclic |= dfsUndirected(neig, u);
        }
        else if (neig != parent) {
            return true;
        }
    }
    return cyclic;
}

/// detect cycles in undirected graph
void detectCycleUndirected() {
    bool isCyclic = false;
    for (int i = 0; i < nodes; ++i) {
        if (!visited[i]) {
            isCyclic |= dfsUndirected(i, i);
        }
    }

    cout << (isCyclic ? "Cyclic\n" : "Acyclic\n");
}
```

```cpp
bool dfsDirected(int u) {
    visited[u] = 1;
    bool cyclic = false;
    for (auto neig: adj_list[u]) {
        if (!visited[neig]) {
            cyclic |= dfsDirected(neig);
        }
        else if (visited[neig] == 1) {
            return true;
        }
    }
    visited[u] = 2;
    return cyclic;
}

/// detect cycles in directed graph
void detectCycleDirected() {
    bool isCyclic = false;
    for (int i = 0; i < nodes; ++i) {
        if (!visited[i]) {
            isCyclic |= dfsDirected(i);
        }
    }

    cout << (isCyclic ? "Cyclic\n" : "Acyclic\n");
}

/// **************************************************************************************************************
*

vi color;

bool dfsBipartite(int u) {
    bool isBipartite = true;
    for (auto neig: adj_list[u]) {
        if (!color[neig]) {
            color[neig] = color[u] == 1 ? 2 : 1;
            isBipartite &= dfsBipartite(neig);
        }
        else if (color[neig] == color[u]) {
            return false;
        }
    }
    return isBipartite;
}

/// check if graph is bipartite
void checkBipartite() {
    bool isBipartite = true;
    for (int i = 0; i < nodes; ++i) {
        if (!color[i]) {
            color[i] = 1;
            isBipartite &= dfsBipartite(i);
        }
    }
    cout << (isBipartite ? "Bipartite\n" : "Not Bipartite\n");
}

void readDFSInput() {
    cin >> nodes >> edges;
    adj_list.resize(nodes);
    visited.assign(nodes, 0);
    color.resize(nodes);

    for (int i = 0; i < edges; ++i) {
        int from, to;
        cin >> from >> to;
        adj_list[from].push_back(to);
        adj_list[to].push_back(from);
    }
}

// GRAPH *******************************************************************************************
void bfs(int src) {
    queue<int> que;
    vi visited(nodes);

    que.push(src);
    visited[src] = 1;
    while (!que.empty()) {
        int u = que.front();
        que.pop();

        cout << u << ' ';
        for (auto v: adj_list[u]) {
            if (!visited[v]) {
                que.push(v);
                visited[v] = 1;
            }
        }
    }
}

void readBFSInput() {
    cin >> nodes >> edges;
    adj_list.resize(nodes);

    for (int i = 0; i < edges; ++i) {
        int from, to;
        cin >> from >> to;
        adj_list[from].push_back(to);
        adj_list[to].push_back(from);
    }
}

vi in_degree;

vi topologicalSort() {
    vi topo_order;
    queue<int> q;

    // push to queue all vertices with in-degree of 0
    for (int i = 0; i < nodes; ++i) {
        if (!in_degree[i]) {
            q.push(i);
        }
    }

    // process vertices with in-degree of 0
    while (!q.empty()) {
        int u = q.front();
        q.pop();

        topo_order.push_back(u);

        for (auto v: adj_list[u]) {
            // we decreased degree by 1, because we deleted u,
            // which is one of in degrees of v
            if (!(--in_degree[v])) {
                q.push(v);
            }
        }
    }

    return topo_order;
}

void sortInTopologicalOrder() {
    cin >> nodes >> edges;
    adj_list.resize(nodes);

    for (int i = 0; i < edges; ++i) {
        int from, to;
        cin >> from >> to;
        adj_list[from].push_back(to);
        in_degree[to]++;
    }

    auto topo_order = topologicalSort();
    if ((int) topo_order.size() != nodes) {
        cout << "The graph has circular dependencies\n";
    }
    else {
        cout << "Topological ordering:\n";
        for (auto vertex: topo_order) {
            cout << vertex << ' ';
        }
    }
}
```

```cpp
void shortestPath(int src) {
    vi dist(nodes, INF);

    // queue to hold the current nodes
    queue<int> q;

    // push the source node to the queue and set its distance to 0
    q.push(src);
    dist[src] = 0;

    while (!q.empty()) {
        // get the front node in the queue and pop it
        int u = q.front();
        q.pop();

        // add the next nodes to the queue
        for (int v: adj_list[u]) {

            // if the node is not visited yet
            if (dist[v] == INF) {
                dist[v] = dist[u] + 1;
                q.push(v);
            }
        }
    }
}

// count connected components
vector<string> grid;
int rows, columns;

int largestConnected;
int cnt;

void countConnected(int r, int c) {
    if (r == -1 || r == rows || c == -1 || c == columns || visited[r][c] || grid[r][c] == '0') {
        return;
    }

    cnt++;
    visited[r][c] = 1;

    countConnected(r, c - 1); // left
    countConnected(r, c + 1); // right
    countConnected(r - 1, c); // up
    countConnected(r + 1, c); // down

    countConnected(r - 1, c - 1); // left-up
    countConnected(r - 1, c + 1); // right-up
    countConnected(r + 1, c - 1); // left-down
    countConnected(r + 1, c + 1); // right-down
}

void solveCountConnected() {
    for (int i = 0; i < rows; ++i) {
        for (int j = 0; j < columns; ++j) {
            if (grid[i][j] == '1') {
                countConnected(i, j);
                largestConnected = max(largestConnected, cnt);
                cnt = 0;
            }
        }
    }
}


// DP ***********************************************************************************************
const int N = 1e3;

ll DP_Template(ll state1, ll state2) {
    // base case
    if () {
        // validate
        // return valid answer
    }

    // memoization
    // -1 is not solution for the problem
    if (dp[state1][state2] != -1) {
        return dp[state1][state2];
    }

    // choices
    ll opt1 = DP_Template();
    ll opt2 = DP_Template();

    // store best solution
    dp[state1][state2] = max(opt1, opt2); // for example

    // return best solution
    return dp[state1][state2];
}

// LIS
vvi dp(N, vi(N, -1));
int n;
vi nums;

int recursiveLIS(int i, int last) {
    if (i == n + 1) {
        return 0;
    }

    int &ret = dp[i][last];
    if (ret != -1) {
        return ret;
    }

    int opt1 = 0;
    if (nums[i] > nums[last]) {
        opt1 = solve(i + 1, i) + 1;
    }

    int opt2 = solve(i + 1, last);

    ret = max(opt1, opt2);
    return ret;
}

// Iterative LIS
void iLIS() {
    dp[0] = 1;
    for (int i = 1; i < n; i++) {
        dp[i] = 1;
        for (int j = 0; j < i; j++)
            if (arr[i] > arr[j] && dp[i] < dp[j] + 1)
                dp[i] = dp[j] + 1;
    }
}

void LCS() {
    for (int i = 0; i <= m; i++) {
        for (int j = 0; j <= n; j++) {
            if (i == 0 || j == 0)
                L[i][j] = 0;
            else if (X[i - 1] == Y[j - 1])
                L[i][j] = L[i - 1][j - 1] + 1;
            else
                L[i][j] = max(L[i - 1][j], L[i][j - 1]);
        }
    }
}

void IterativeLIS_Diff1() {
    for (int i = 0; i < n; ++i) {
        cin >> v[i];
        mp[v[i]] = 0;
    }

    for (int i = 0; i < n; ++i) {
        dp[v[i]] = max(dp[v[i]], dp[v[i] - 1] + 1);
    }

    int mx = 0, value = 0;
    for (auto x: mp) {
        if (x.second > mx) {
            mx = x.second;
            value = x.first;
        }
    }

    vector<ll> ans;
    for (int i = n - 1; i >= 0; --i) {
        if (v[i] == value) {
            ans.push_back(i + 1);
            value--;
        }
    }

    reverse(ans);
}

void knapsack1D() {
    for (int i = 0; i < n; i++) {
        for (int j = W; j >= wt[i]; j--) {
            dp[j] = max(dp[j], val[i] + dp[j - wt[i]]);
        }
    }
}

void knapsack2D() {
    for (int i = 0; i < N; ++i) {
        for (int j = 0; j <= W; ++j) {
            if (j + w[i] <= W) {
                dp[i][j] = max(dp[i + 1][j + w[i]], dp[i][j] + v[i]);
            }
        }
    }
}

void knapsackRecursive() {
    // from main, pass (0, capacity)
    int solve(int i, int rem) {
        if (i == n + 1 || rem == 0) {
            return 0;
        }

        if (dp[i][rem] != -1) {
            return dp[i][rem];
        }

        // pick current item
        int option1;
        if (w[i] <= rem) {
            option1 = v[i] + solve(i + 1, rem - w[i]);
        }
        else {
            option1 = 0;
        }

        // leave the current item
        int option2 = solve(i + 1, rem);

        dp[i][rem] = max(option1, option2);
        return dp[i][rem];
    }
// BITMASK ********************************************************************************
    void bitMask() {
        for (int mask = 0; mask < (1 << n); mask++) {

            int sum = 0;
            for (int i = 0; i < n; i++) {
                if ((mask >> i) & 1)
                    sum += nums[i];

            }

            if (sum == target)
                counter++;
        }
    }
```