🔍 How to Identify Graph Input in Competitive Programming Problems

Whenever you see input like:

```
n m
u₁ v₁
u₂ v₂
...
uₘ vₘ
```

You should immediately think: ☑ **This is a graph**.

🔹 What to check in the problem statement:

1. **Does it say "directed" or "undirected"?**
   - ➤ If **undirected**, add both `u → v` and `v → u`.
   - ➤ If **directed**, add only `u → v`.
2. **Are there weights on edges?**
   - ➤ If yes, the input will be: `u v w`
   - ➤ In that case, use: `vector<pair<int, int>> adj[];`
3. **Is indexing 0-based or 1-based?**
   - ➤ If nodes are numbered from `1 to n`, use `adj[n+1]`
   - ➤ If from `0 to n-1`, use `adj[n]`

🧠 When to Use Matrix vs List

Representation	Use When…	Space	Time to Find Neighbor
Adjacency Matrix	Graph is **dense** (`m ≈ n²`) or you need fast edge lookup	$O(n^2)$	$O(1)$
Adjacency List	Graph is **sparse** (`m << n²`)	$O(n + m)$	O(#neighbors)

🧰 Template Recognition Based on Input

Here's a cheat-sheet based on **problem input patterns**:

➤ Unweighted Undirected Graph

```
Input:
n m
u₁ v₁
```

```
u₂ v₂
...
```

- Representation:

```
vector<int> adj[n+1];
adj[u].push_back(v);
adj[v].push_back(u);
```

➤ Unweighted Directed Graph

```
Input:
n m
u₁ v₁
...
```

- Representation:

```
vector<int> adj[n+1];
adj[u].push_back(v); // no reverse
```

➤ Weighted Undirected Graph

```
Input:
n m
u v w
```

- Representation:

```
vector<pair<int, int>> adj[n+1];
adj[u].push_back({v, w});
adj[v].push_back({u, w});
```

➤ Weighted Directed Graph

```
Input:
n m
u v w
```

- Representation:

```
vector<pair<int, int>> adj[n+1];
adj[u].push_back({v, w}); // no reverse
```

Weighted Undirected Graph

🕹️ **Input:**

```
n m
u v w   // edge between u and v with weight w
```

📦 Code:

```
vector<pair<int, int>> adj[n+1];  // 1-based indexing

for (int i = 0; i < m; i++) {
    int u, v, w;
    cin >> u >> v >> w;
    adj[u].push_back({v, w});
    adj[v].push_back({u, w}); // because undirected
}
```

🧭 Weighted Directed Graph

📦 Code:

```
vector<pair<int, int>> adj[n+1];

for (int i = 0; i < m; i++) {
    int u, v, w;
    cin >> u >> v >> w;
    adj[u].push_back({v, w}); // only one direction
}
```

☑ 2. Using `vector<vector<int>> adj[]`

This is **less common** but also possible.

🔹 Structure:

Each index of `adj[u]` contains a list of **vector<int>**, where each vector holds two values:

```
adj[u].push_back({v, w});
```

- `v` → destination node
- `w` → weight

🧭 Weighted Undirected Graph

📦 Code:

```
vector<vector<int>> adj[n+1];  // 1-based indexing

for (int i = 0; i < m; i++) {
    int u, v, w;
    cin >> u >> v >> w;
```

```
    adj[u].push_back({v, w});
    adj[v].push_back({u, w});
}
```

🧭 Weighted Directed Graph

📦 Code:

```
vector<vector<int>> adj[n+1];

for (int i = 0; i < m; i++) {
    int u, v, w;
    cin >> u >> v >> w;
    adj[u].push_back({v, w});
}
```

🆚 Comparison: `vector<pair<int,int>>` vs `vector<vector<int>>`

Aspect	`vector<pair<int, int>>`	`vector<vector<int>>`
Preferred for readability	☑ Yes	❌ Slightly less readable
Easier to access elements	☑ `it.first`, `it.second`	❌ `it[0]`, `it[1]` (less clear)
Syntax	`adj[u].push_back({v, w})`	`adj[u].push_back({v, w})`
Used in real-world problems	☑ Commonly used in Dijkstra, MST	⚠️ Less common

📘 What is an Adjacency Matrix?

An **adjacency matrix** is a `2D array` (or matrix) of size `n x n` where `n = number of nodes`.

- `matrix[i][j] = 1` (or weight `w`) if there is an edge from `i` to `j`.
- `matrix[i][j] = 0` if there is no edge between them.

🧭 Input Format (Commonly used in contests & problems):

```
n m
u v         // for unweighted graph
```

```
or
u v w       // for weighted graph
```

- `n` = number of nodes
- `m` = number of edges
- `u`, `v` = nodes
- `w` = weight of edge (only for weighted)

☑ 1. Unweighted Undirected Graph

```
int adj[n+1][n+1] = {0};

for (int i = 0; i < m; i++) {
    int u, v;
    cin >> u >> v;
    adj[u][v] = 1;
    adj[v][u] = 1; // undirected: add both ways
}
```

Example:

```
Input:
5 3
1 2
2 3
4 5
```

Matrix:

```
CopyEdit
  1 2 3 4 5
1 0 1 0 0 0
2 1 0 1 0 0
3 0 1 0 0 0
4 0 0 0 0 1
5 0 0 0 1 0
```

☑ 2. Unweighted Directed Graph

```
adj[u][v] = 1; // Only one direction
```

- Do **not** set `adj[v][u] = 1`.

☑ 3. Weighted Undirected Graph

```
int adj[n+1][n+1];
```

```
for (int i = 0; i < m; i++) {
    int u, v, w;
    cin >> u >> v >> w;
    adj[u][v] = w;
    adj[v][u] = w; // undirected
}
```

☑ 4. Weighted Directed Graph

```
adj[u][v] = w; // directed: only one direction
```

💡 Notes:

- Adjacency matrix takes $O(n^2)$ space.
- Good for **dense graphs** or when you want **quick edge checks**: $O(1)$ time to check if edge exists.
- Not space-efficient for **sparse graphs** (use adjacency list for that).

📃 Printing the Matrix:

```
for (int i = 1; i <= n; i++) {
    for (int j = 1; j <= n; j++) {
        cout << adj[i][j] << " ";
    }
    cout << "\n";
}
```

🔎 Final Clues That the Problem is a Graph

Watch out for phrases like:

- “There are n nodes and m connections”
- “u is connected to v”
- “Move from one node to another”
- “Reachable”
- “Find path, cycle, component, distance, etc.”
- “Task depends on another”
- “Grid where you can move in directions”

These are all **graph concepts**, and the input will match one of the above formats.