Graph Travserals and Bipartite Graphs

- Gaurish Baliga

Goal

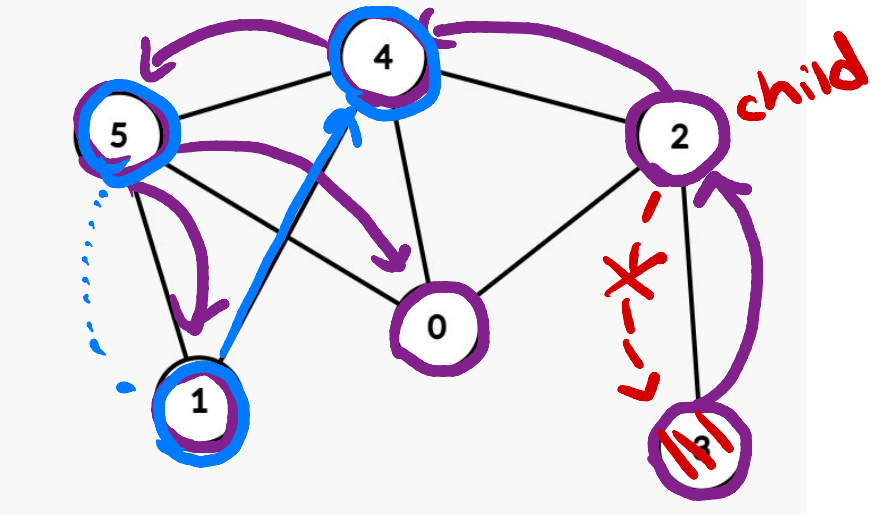
- Depth First Search ✓
- Breadth First Search ✓
- Problems on Traversals ✓
- Grids as Graphs → *
- Bipartite Graphs → cool!

Depth First Search

In Depth First Search (or DFS) for a graph, we traverse all adjacent vertices one by one. When we traverse an adjacent vertex, we completely finish the traversal of all vertices reachable through that adjacent vertex.

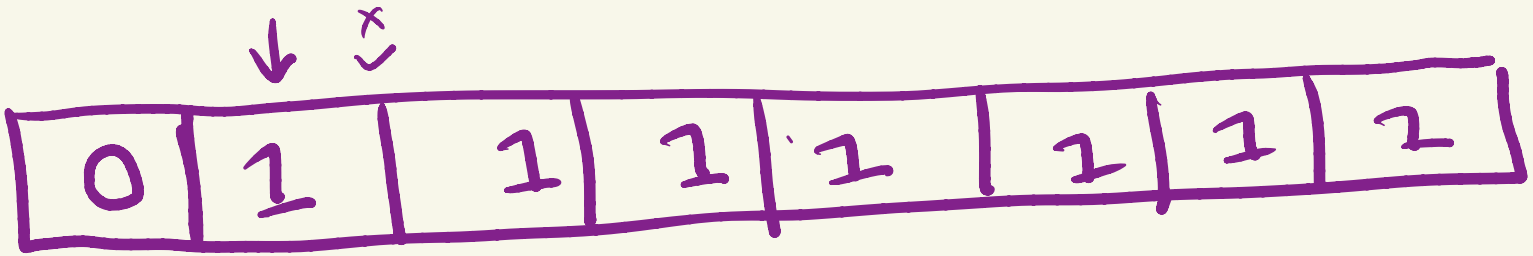
Note: DFS Traversal is not unique for a graph, multiple traversal orders exist

Depth First Search Example



Tree s dfs (—, par)

DFS Traversal : [3, 2, 4, 5, 1, 0]

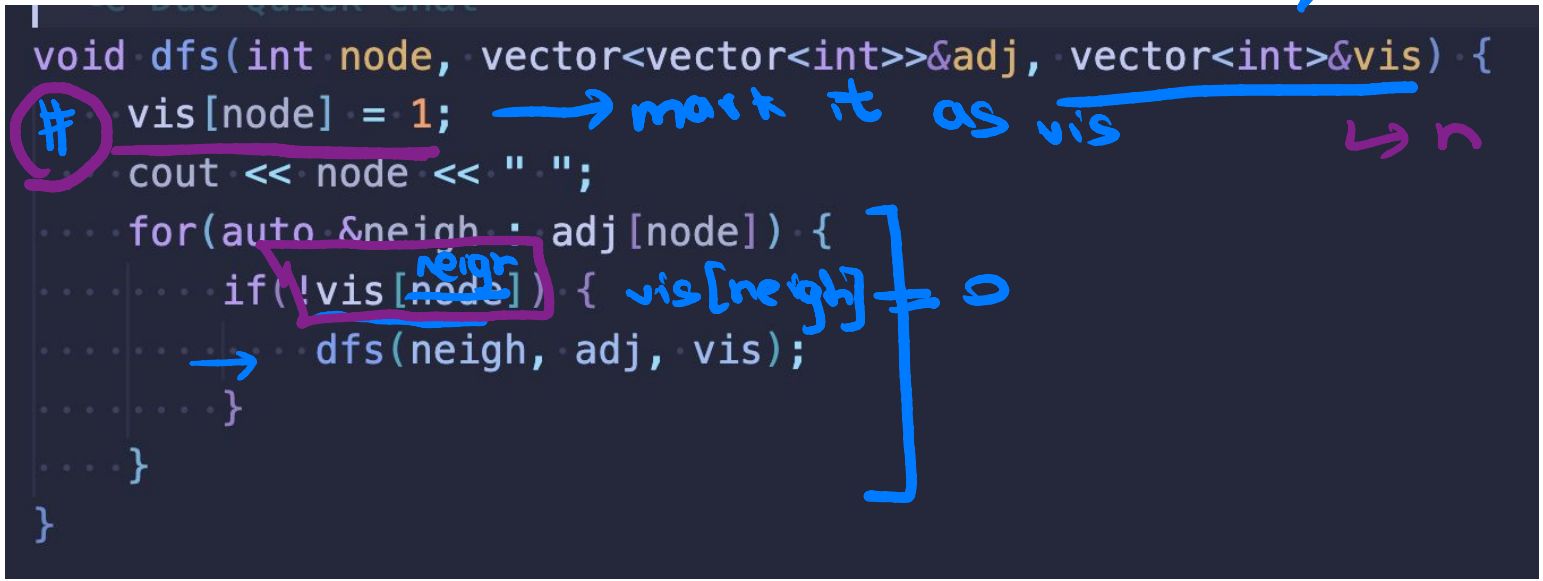


Visited array : vis[i]

↳ 0

↳ 1 node i is vis

Depth First Search Implementation



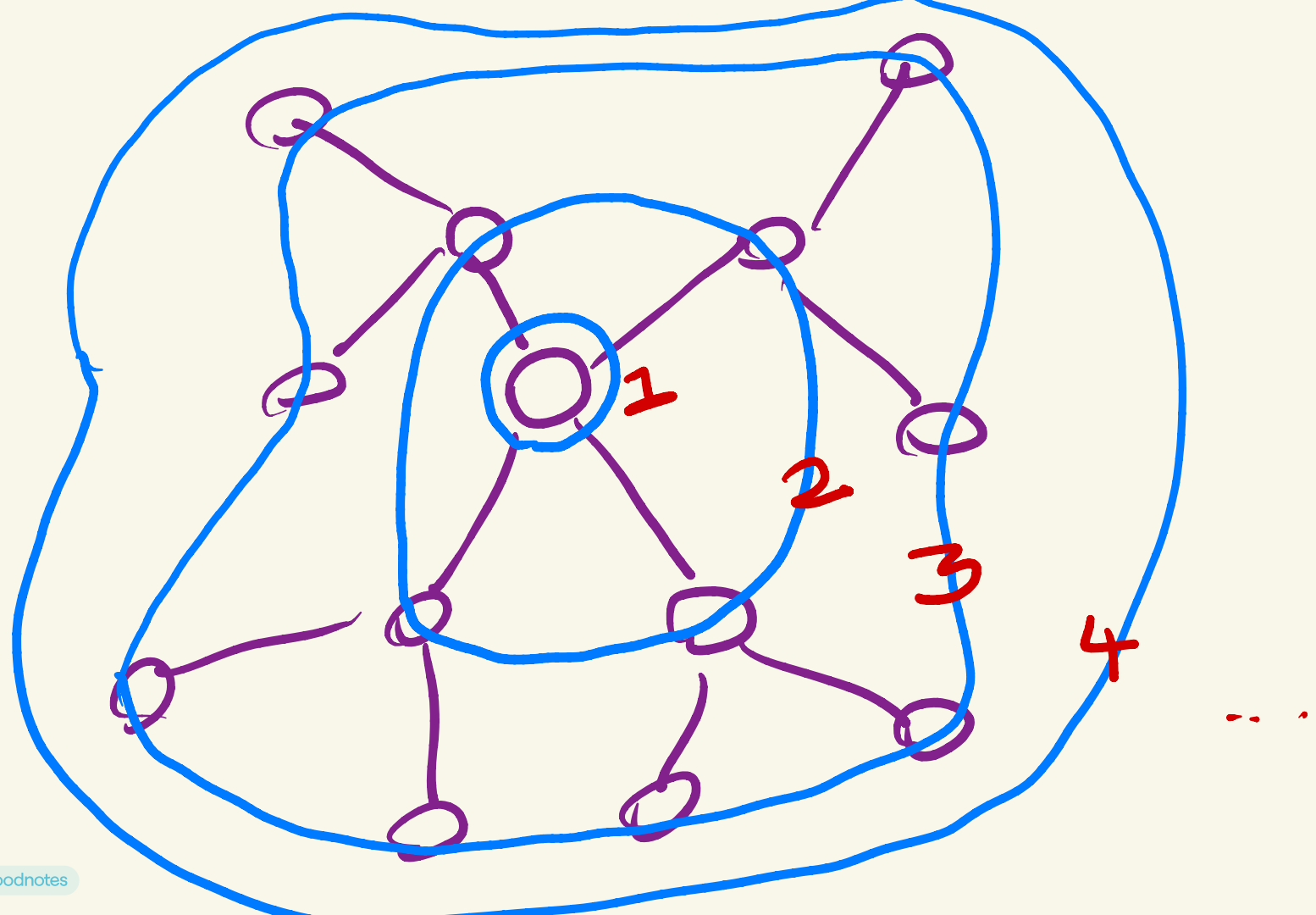
— : changes from trees

Breadth First Search

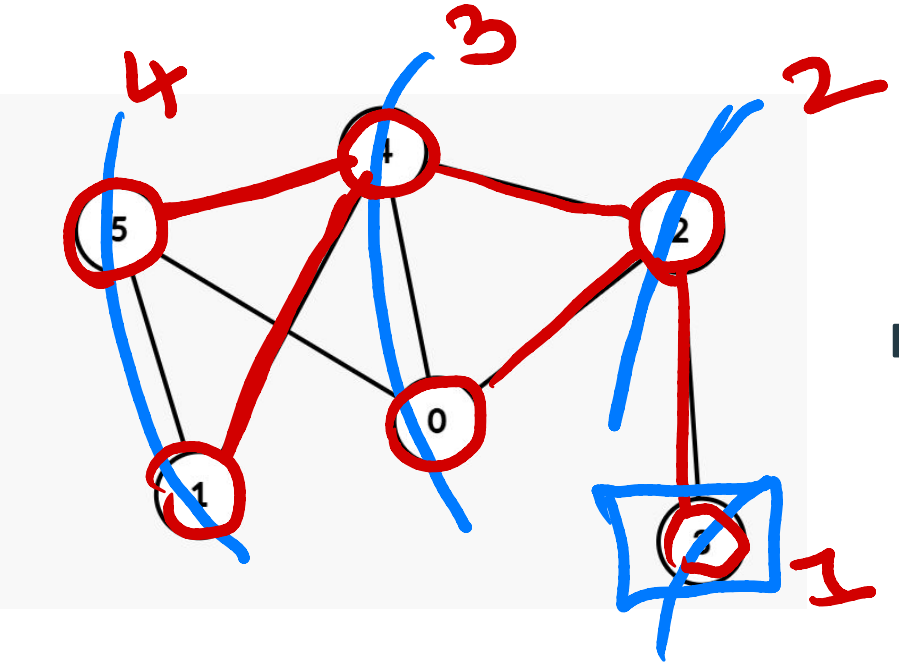
Breadth First Search (BFS) expands its breadth one by one. It first visits an entire level, then visits the next entire level and so on.

We use a queue to implement the traversal since we want to visit all nodes in a level and then its children and so on, so forth.

Note: Just like DFS, BFS Traversal is not unique for a given graph.



Breadth First Search Example

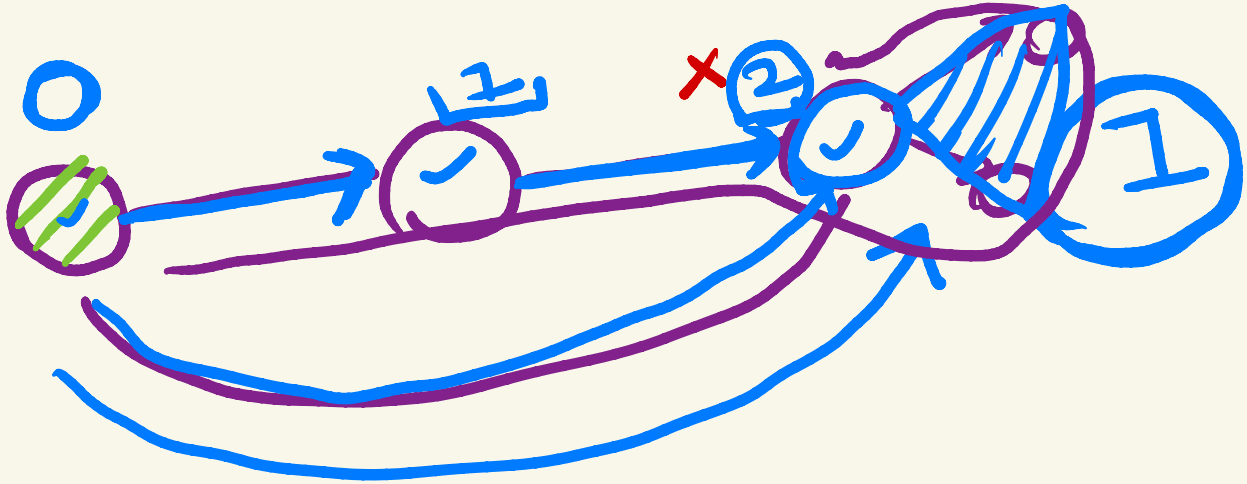


BFS Traversal: [3, 2, 0, 4, 5, 1]

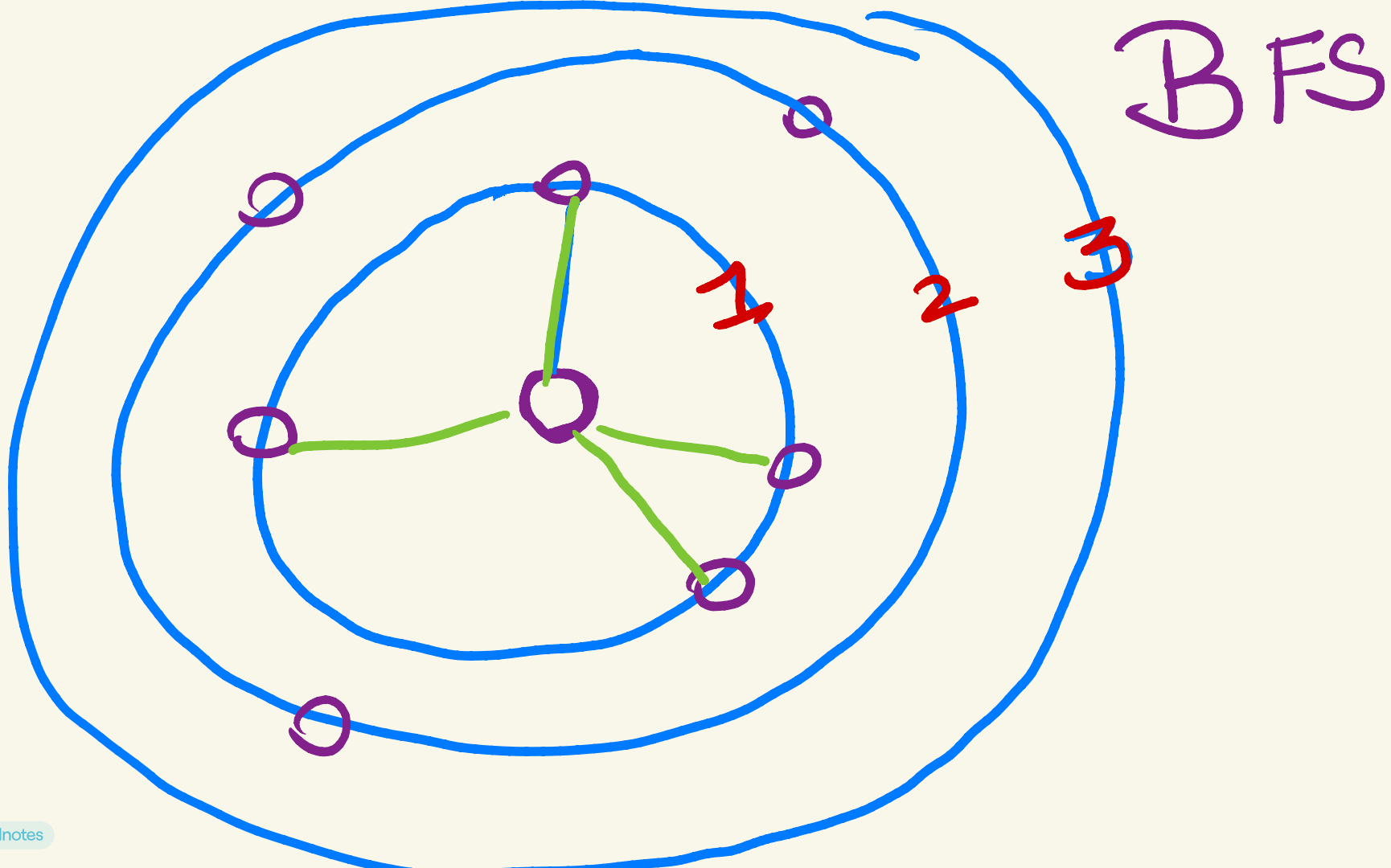
Consider an undirected unweighted graph.

given

→ shortest distance from (s) to all other nodes

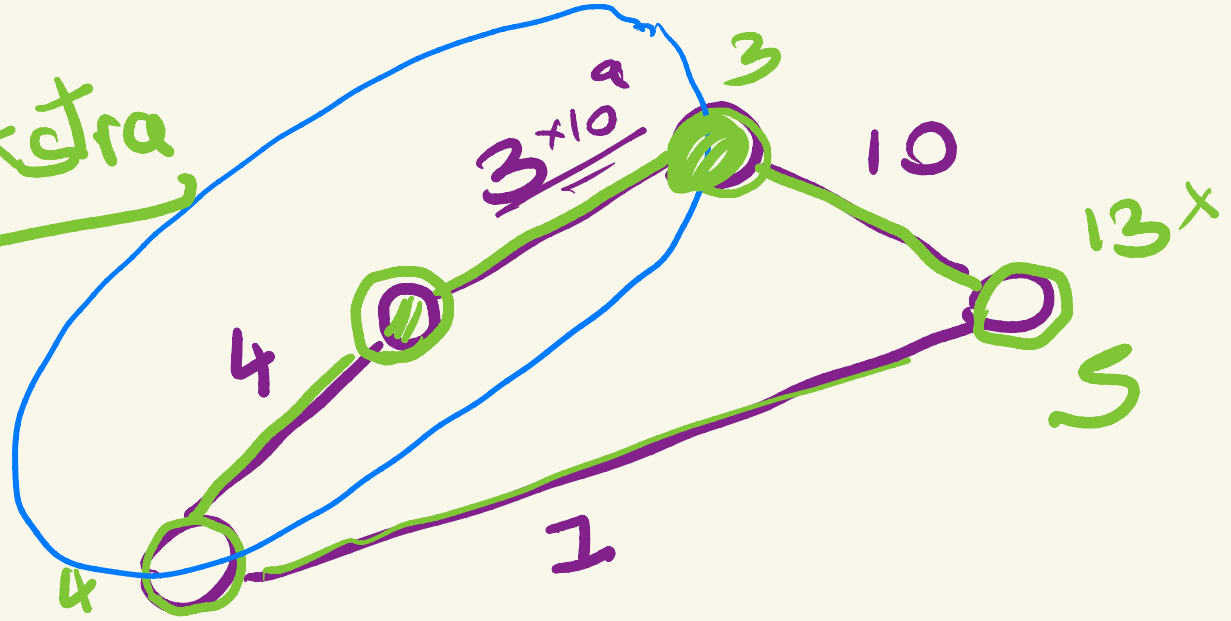


Why DFS fails?

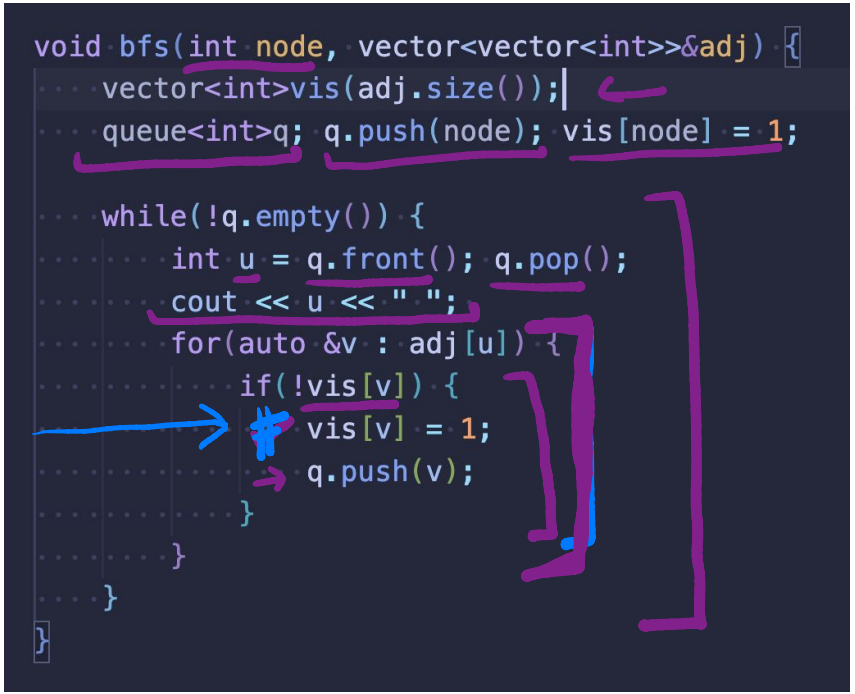


Why not weighted graphs as well??

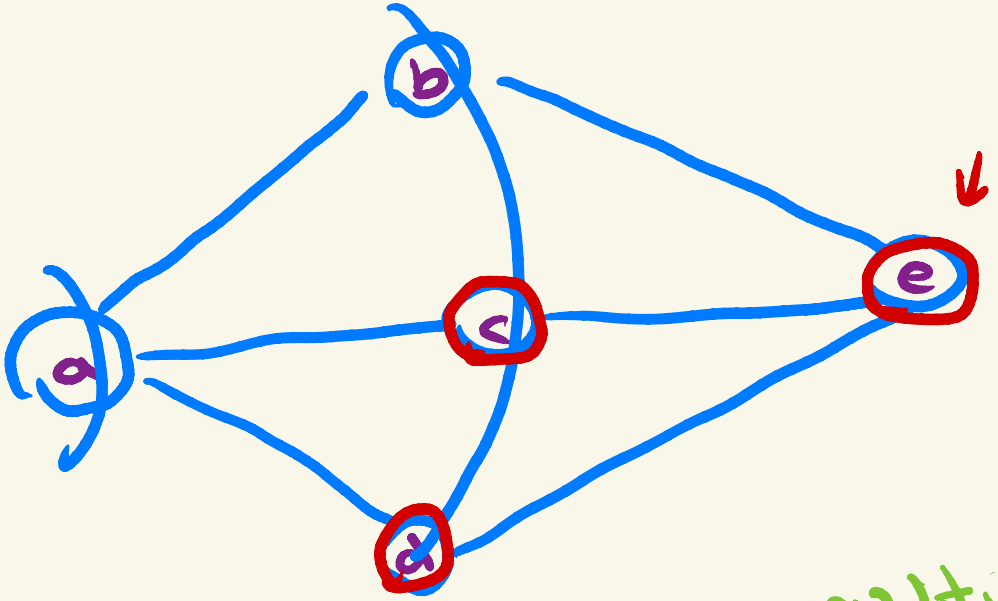
↳ Dijkstra



Breadth First Search Implementation



Why not mark visited later?



0 / not visited

multiple times

{ a, b, c, d, e, e, e }

Time complexity: $O(n+m)$

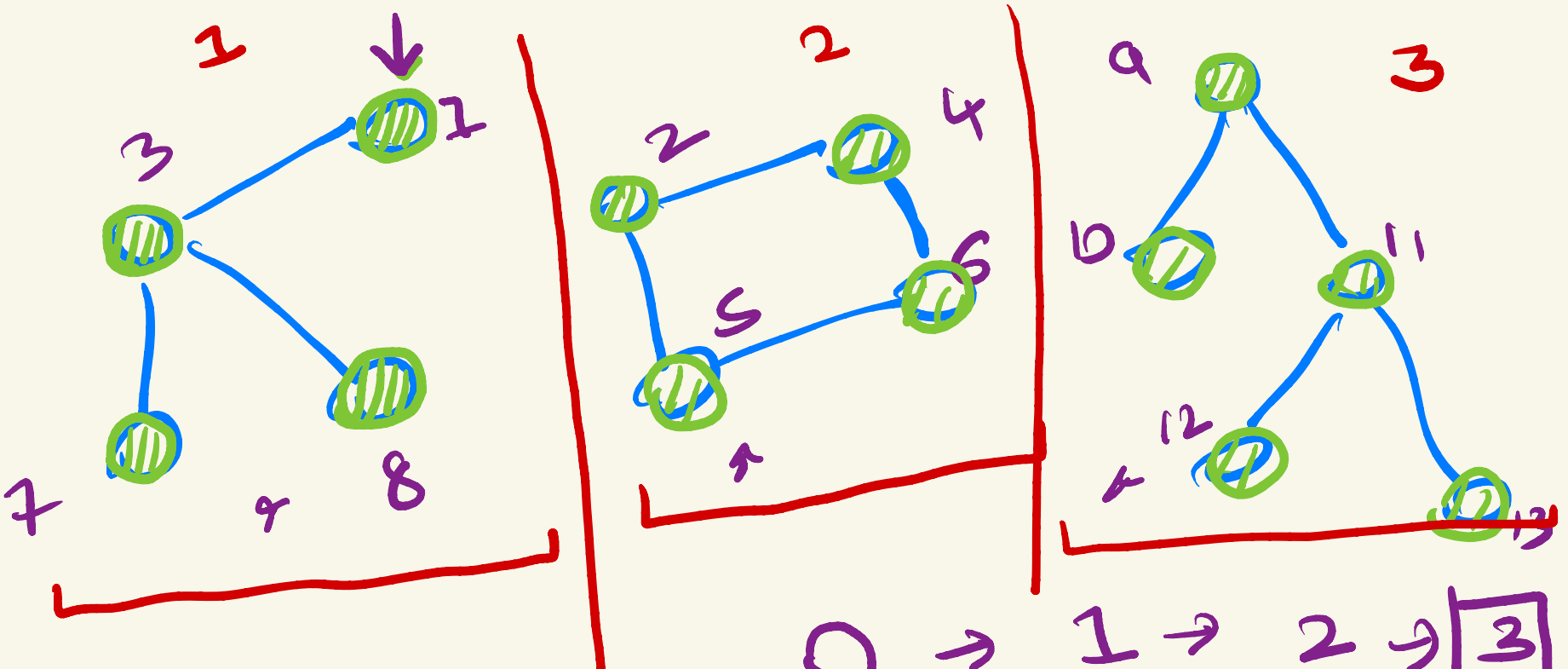
nodes edges

Space complexity: $O(n)$

1000 nodes 10^6 edges

Problems to Consider:

1. Given a Graph, Find the Number of Connected Components.
2. Given a Graph, Find the Size of each Connected Component.
3. Minimum Time to Reach each cell from a source node in an unweighted graph. → BFS



0 → 1 → 2 → 3

3

no. of connected comp.

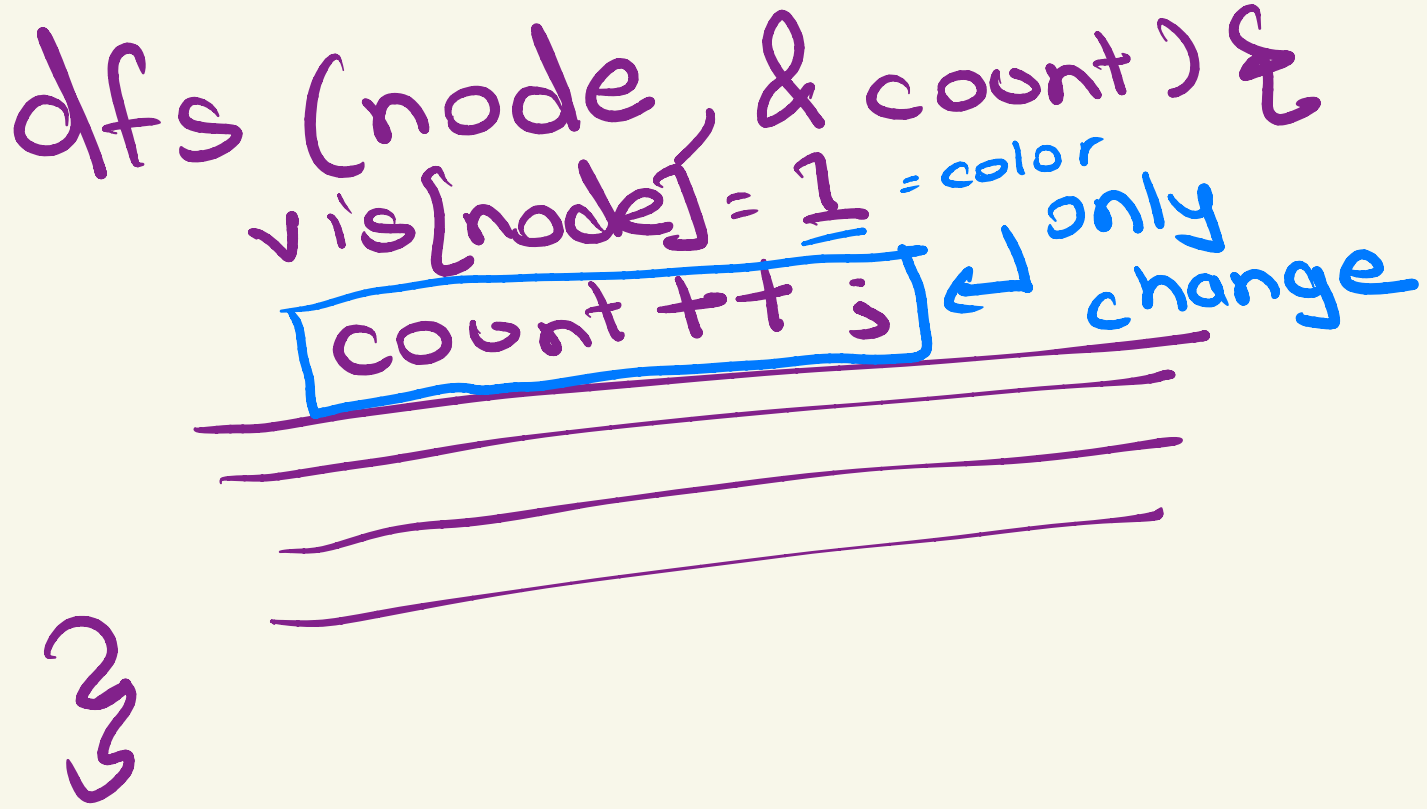
```
int count = 0;

for (int i = 1; i <= n; i++) {
    if (!vis[i]) {
        count++;
        -> dfs(i);
    }
}
```

O(n+m) ✓

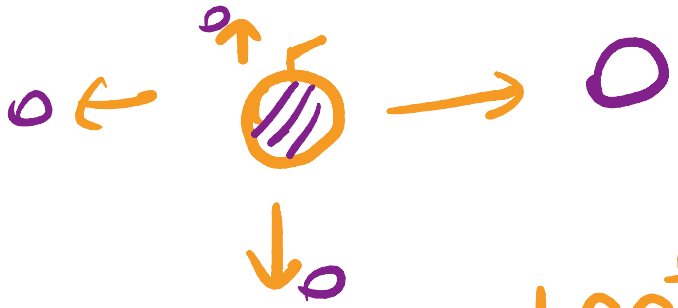
Printing size of each cc.

```
for (int i = 1; i <= n; i++) {
    if (!vis[i]) {
        int count = 0;
        → dfs(i, count);
        cout << count << " ";
    }
}
```

color



Graphs as Grids

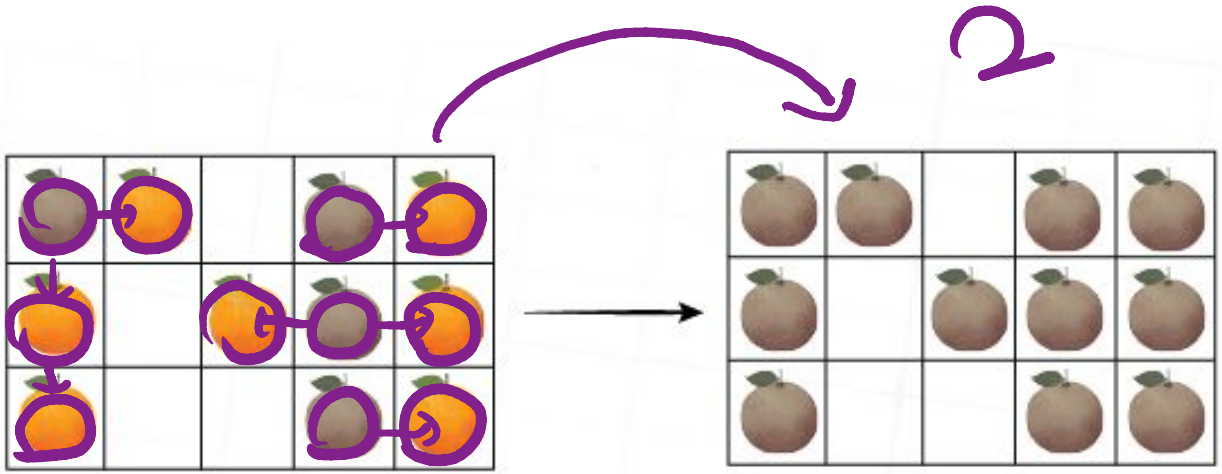


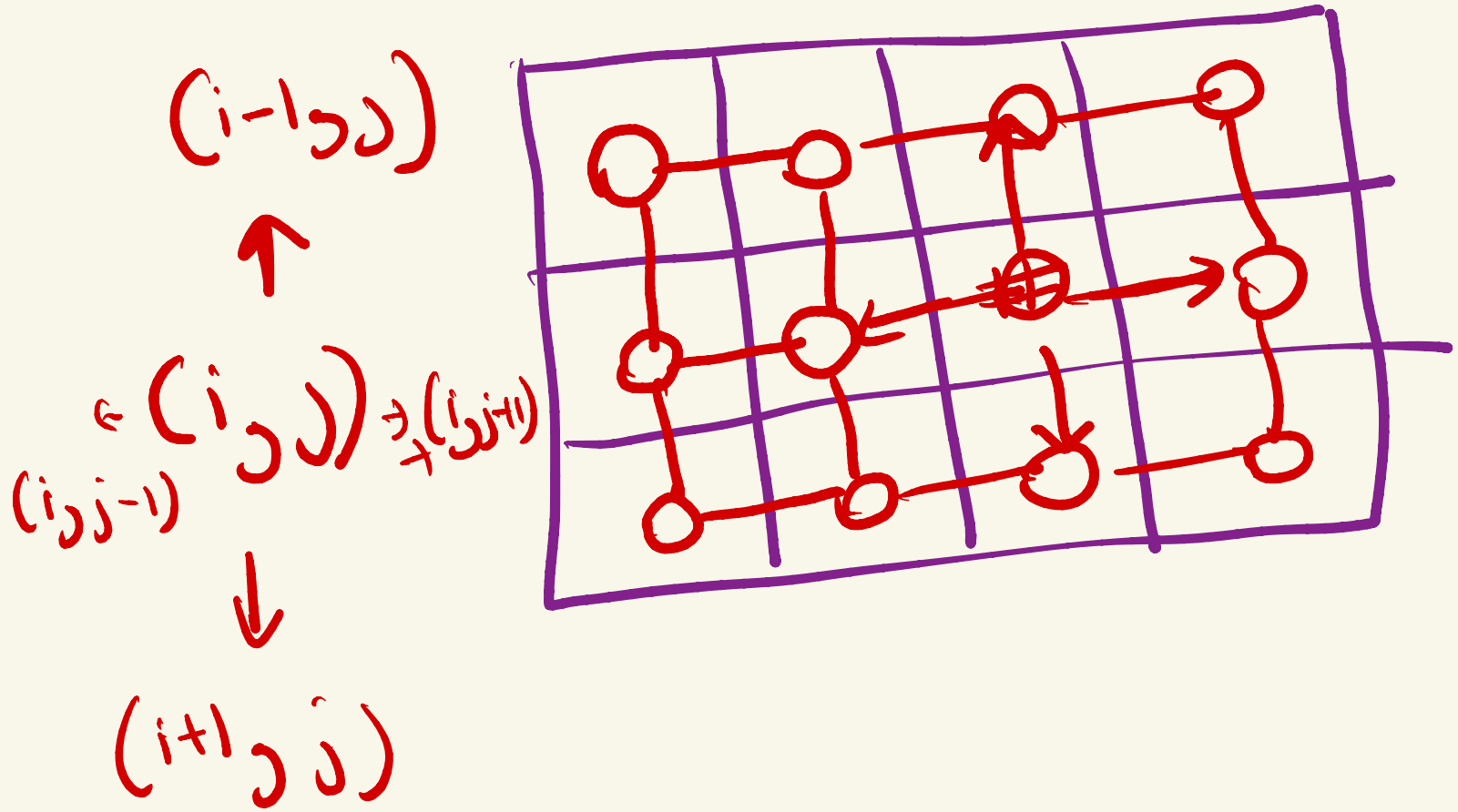
Rotten oranges Leetcode

Consider this problem:

You are given an n * m grid where each cell is initially empty, or has a fresh or rotten orange. Each rotten orange every second infects its neighbouring oranges and they also become rotten. What is the minimum time required to make all oranges rotten in the grid

Example: (Answer is 2 seconds)





Correlation Between Graphs and Grids

You can notice that a grid, is a just a graph where cell (i, j) is adjacent with cells (i - 1, j), (i + 1, j), (i, j - 1), (i, j + 1).

Now the task is just given some “special” nodes on the graph, to find the minimum time to visit all nodes via this “special” nodes.

We can use BFS, but insert all “special” nodes in the queue.

Concept of DxDy Arrays

```
Int dx[] = {-1, 1, 0, 0};
Int dy[] = {0, 0, 1, -1};
```

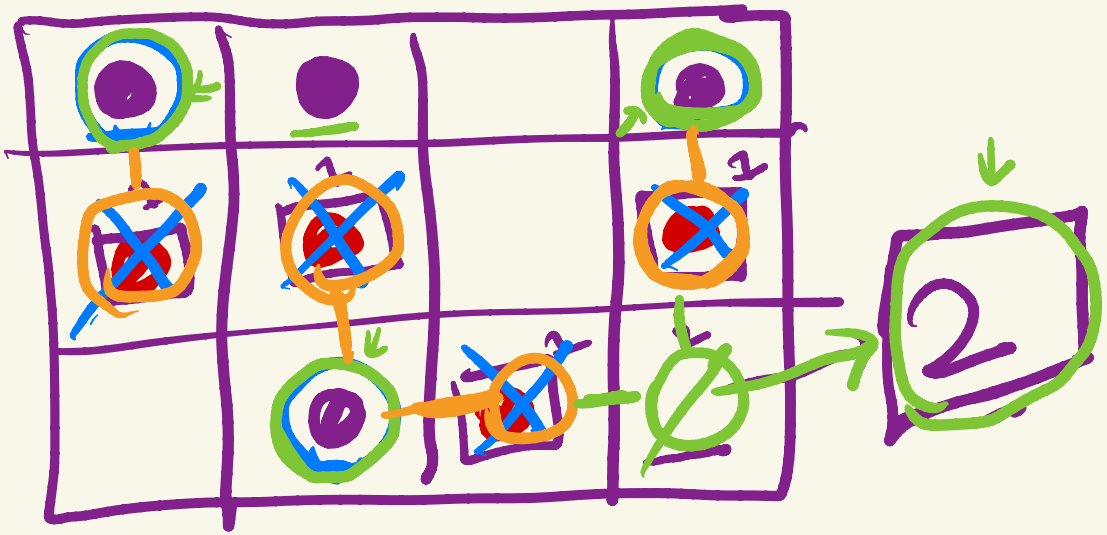
```
for(i=0; i<4; i++){
    newX = x + dx[i]
    newY = y + dy[i]
}
```

If you iterate over indices i from 0, 1, 2, 3 and add dx[i] to x and dy[i] to y, you will notice that you are iterating over the adjacent cells of a cell in a grid.

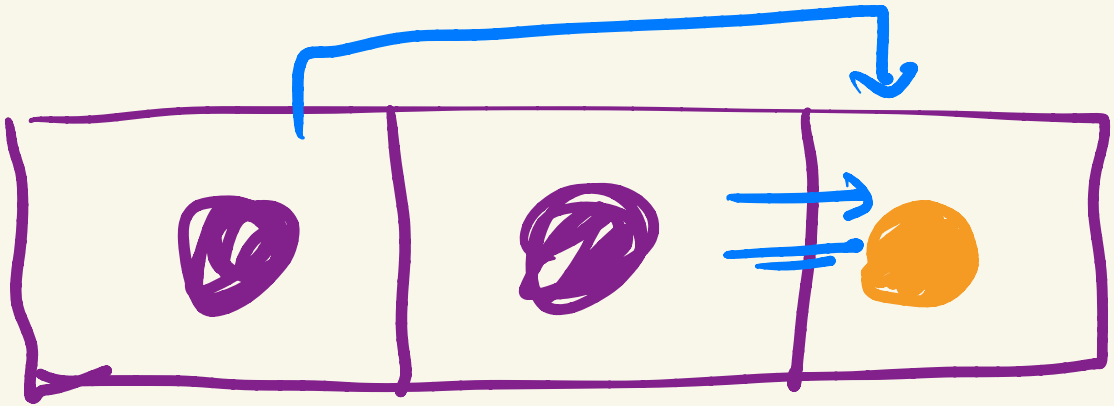
For Example for **(x, y) : (x - 1, y), (x + 1, y), (x, y + 1), (x, y - 1)**

Find max min distance of an orange from rotten

$n \times m \leq 10^6$



$O(n^2 m^2)$ → Multisource BFS → $O(n \cdot m)$



Brute Force : ✗ Wont Work TLE

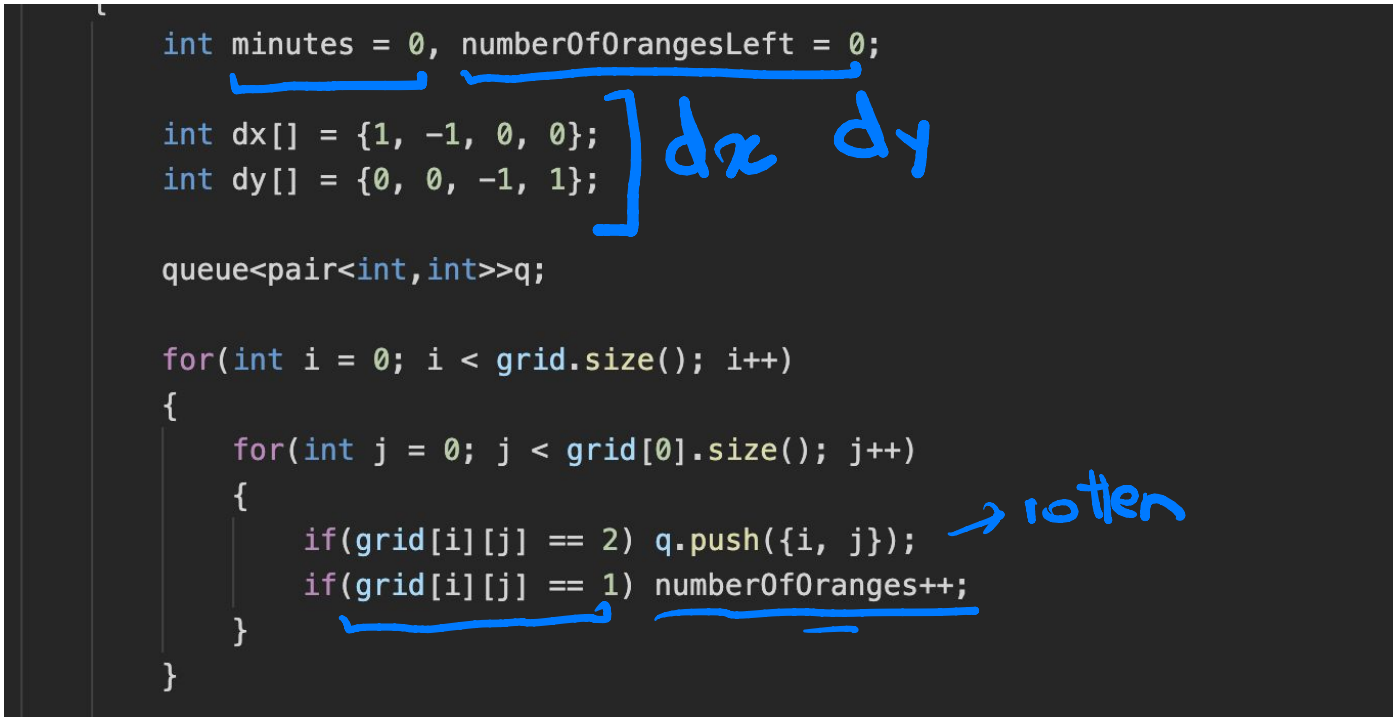
For each rotten orrange do BFS call

$O(n \cdot m)$ → $O(n \cdot m)$ BFS

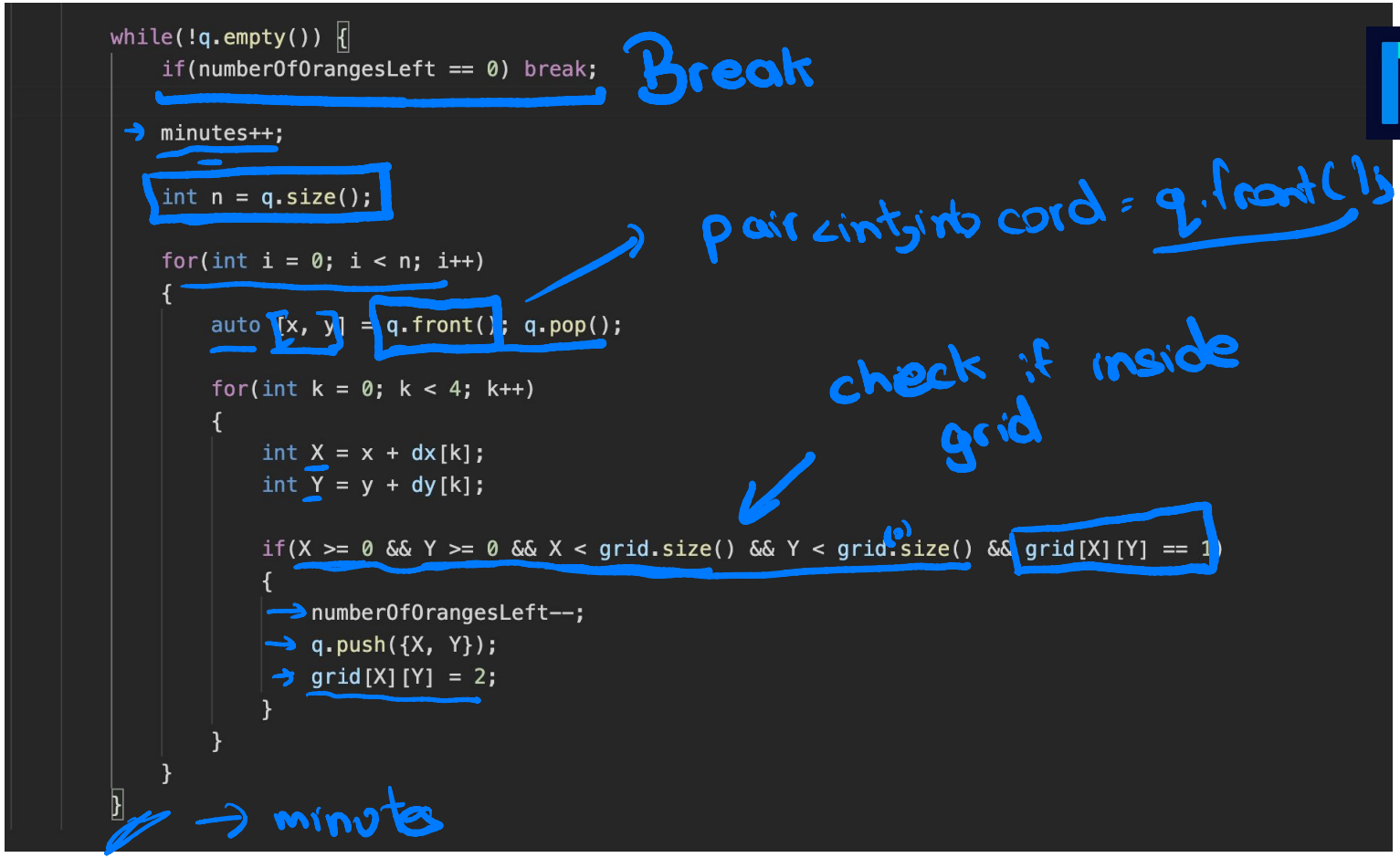
$\underbrace{\quad\quad}_{\text{[rotten or]}}$

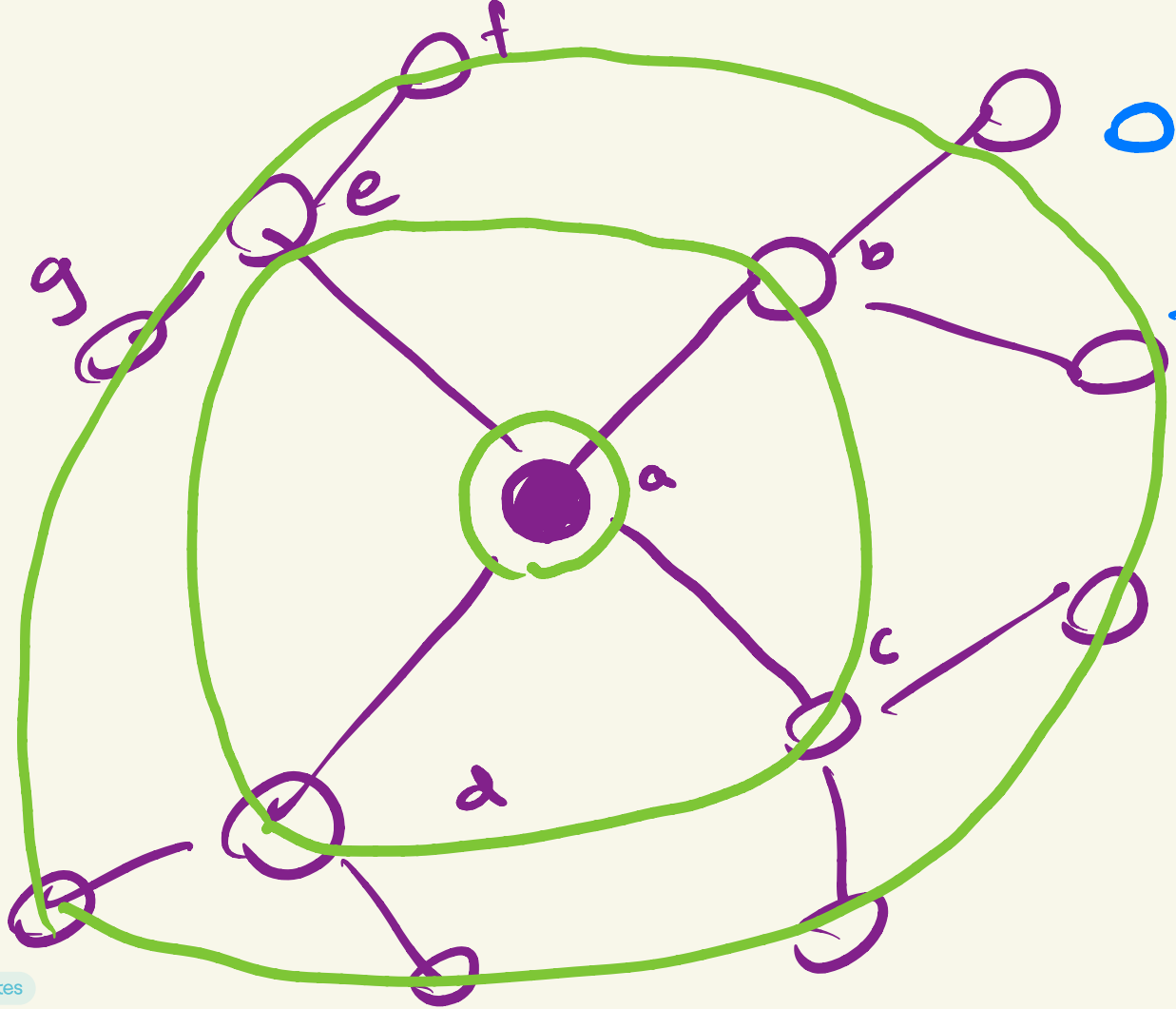
Multisource BFS:

Implementation (Continued on Next Page)



```
{
    int minutes = 0, numberOfOrangesLeft = 0;

    int dx[] = {1, -1, 0, 0};
    int dy[] = {0, 0, -1, 1};

    queue<pair<int,int>>q;

    for(int i = 0; i < grid.size(); i++)
    {
        for(int j = 0; j < grid[0].size(); j++)
        {
            if(grid[i][j] == 2) q.push({i, j});
            if(grid[i][j] == 1) numberOfOranges++;
        }
    }
```


while(!q.empty()) {
if(numberOfOrangesLeft == 0) break;
Break
minutes++;
int n = q.size();
pair<int,int> cord = q.front();
for(int i = 0; i < n; i++)
{
auto [x, y] = q.front(); q.pop();
for(int k = 0; k < 4; k++)
{
int X = x + dx[k];
int Y = y + dy[k];
check if inside grid
if(X >= 0 && Y >= 0 && X < grid.size() && Y < grid.size() && grid[X][Y] == 1)
{
numberOfOrangesLeft--;
q.push({X, Y});
grid[X][Y] = 2;
}
}
}
}
→ minutes



0 ξ a ,

1 b, c, d, e ,

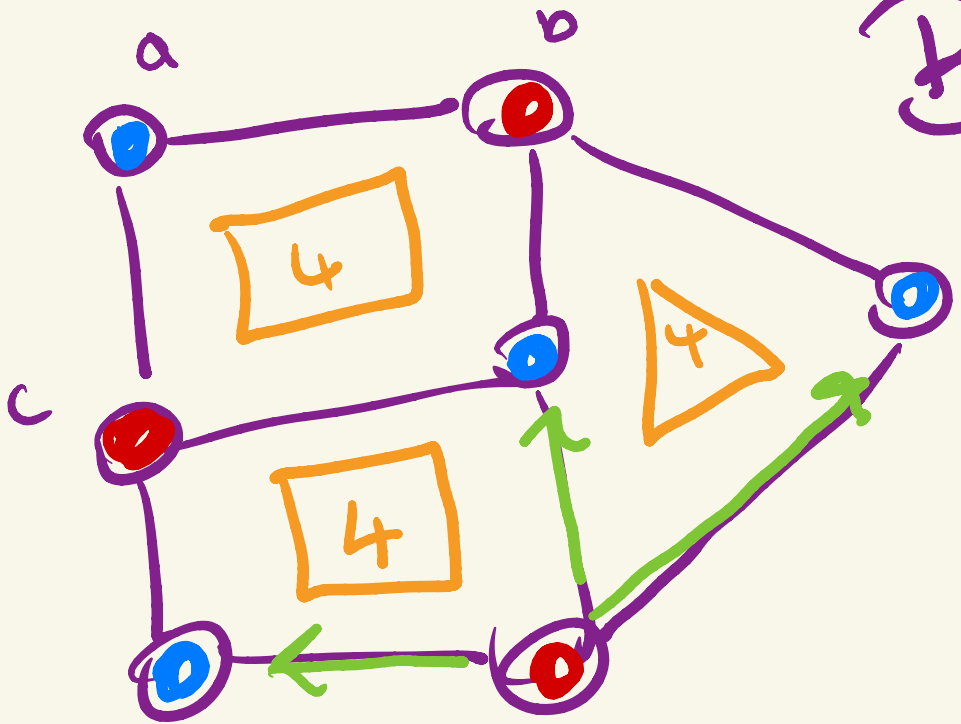
2 f, g, h, ...

3

Bipartite Graphs

A graph is said to be bipartite, if and only if you colour the nodes with white or black and in the end for every nodes, it’s neighbour is collected in a different color.

Bipartite Graph



Check if bipartite or not

Proof :

3 nodes
5 nodes

4 nodes

Tree is always Bipartite

even len/ no cycles → Always Bipartite
odd cycle → not bipartite

node → all adj should be diff color. bool isB...

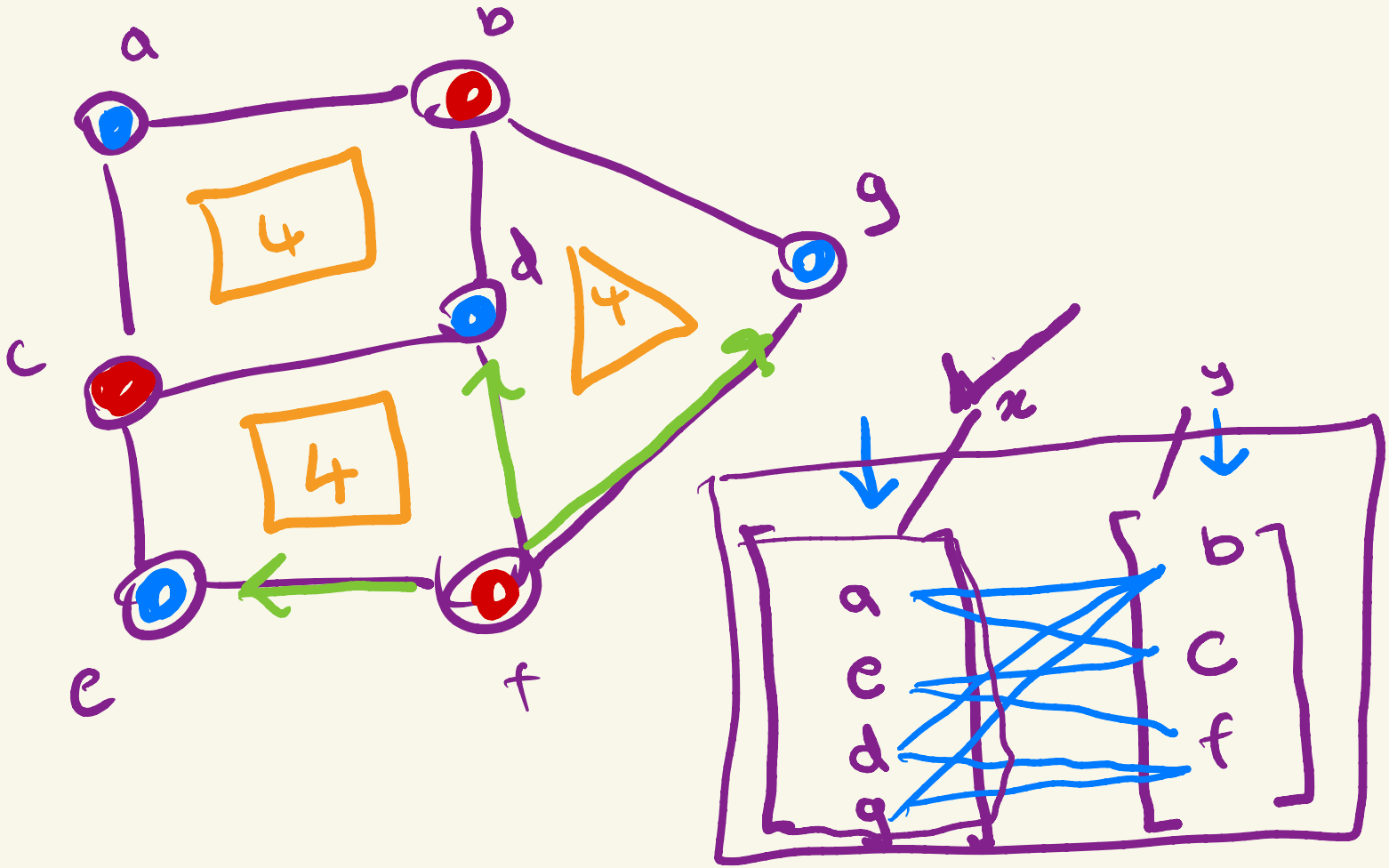
```
dfs (node, color) {
    vis[node] = color.
    for (v : adj[node]) {
        if (vis[v] == 0) dfs(v, otherColor);
        else {
            if (vis[v] ≠ otherColor)
                isBipartite = false.
        }
    }
}
```

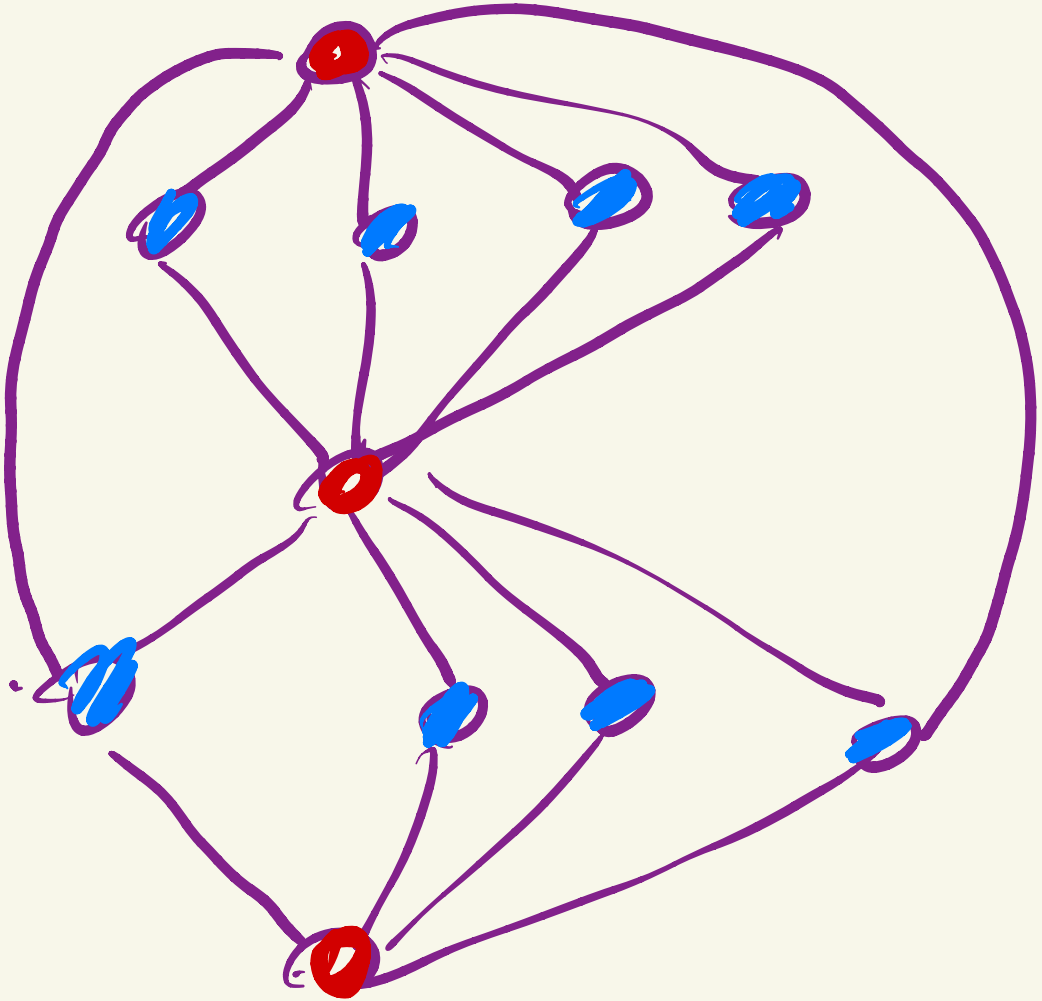
Check if graph has odd length cycle.

↳ Checking if graph is Bipartite

Important Observation

- Bipartite Graphs do not have odd length Cycles
- Bipartite Graphs can be split exhaustively into 2 sets, where each set contains nodes of a particular color. Then edges in the graph will only go from one set to the other set.





Problem:

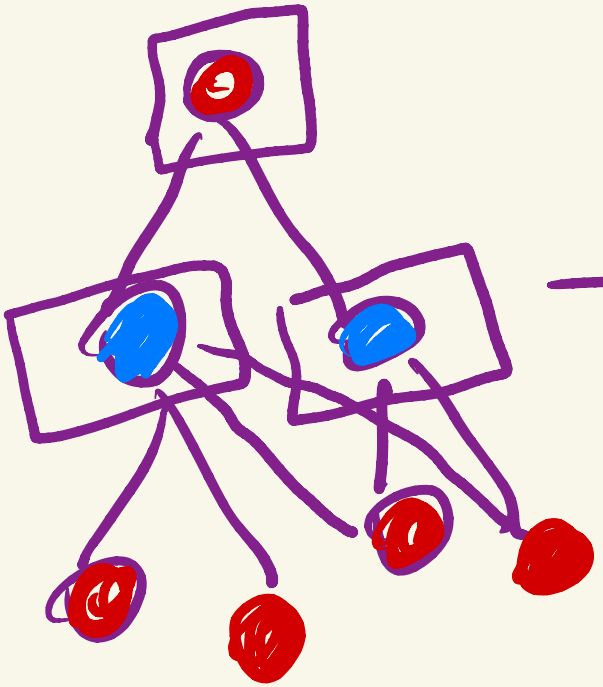
https://codeforces.com/problemset/problem/862/b

Mahmoud and Ehab continue their adventures! As everybody in the evil land knows, Dr. Evil likes bipartite graphs, especially trees.

A tree is a connected acyclic graph. A bipartite graph is a graph, whose vertices can be partitioned into 2 sets in such a way, that for each edge (u, v) that belongs to the graph, u and v belong to different sets. You can find more formal definitions of a tree and a bipartite graph in the notes section below.

Dr. Evil gave Mahmoud and Ehab a tree consisting of n nodes and asked them to add edges to it in such a way, that the graph is still bipartite. Besides, after adding these edges the graph should be simple (doesn't contain loops or multiple edges). What is the maximum number of edges they can add?

A loop is an edge, which connects a node with itself. Graph doesn't contain multiple edges when for each pair of nodes there is no more than one edge between them. **A cycle and a loop aren't the same** .



Max number of edges you can add ?!

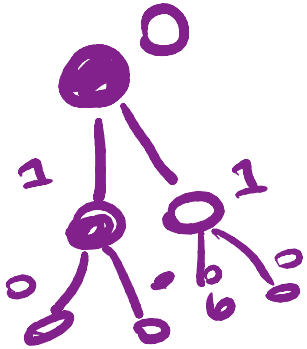
blue ↓ x · red ↓ y $- (n-1)$

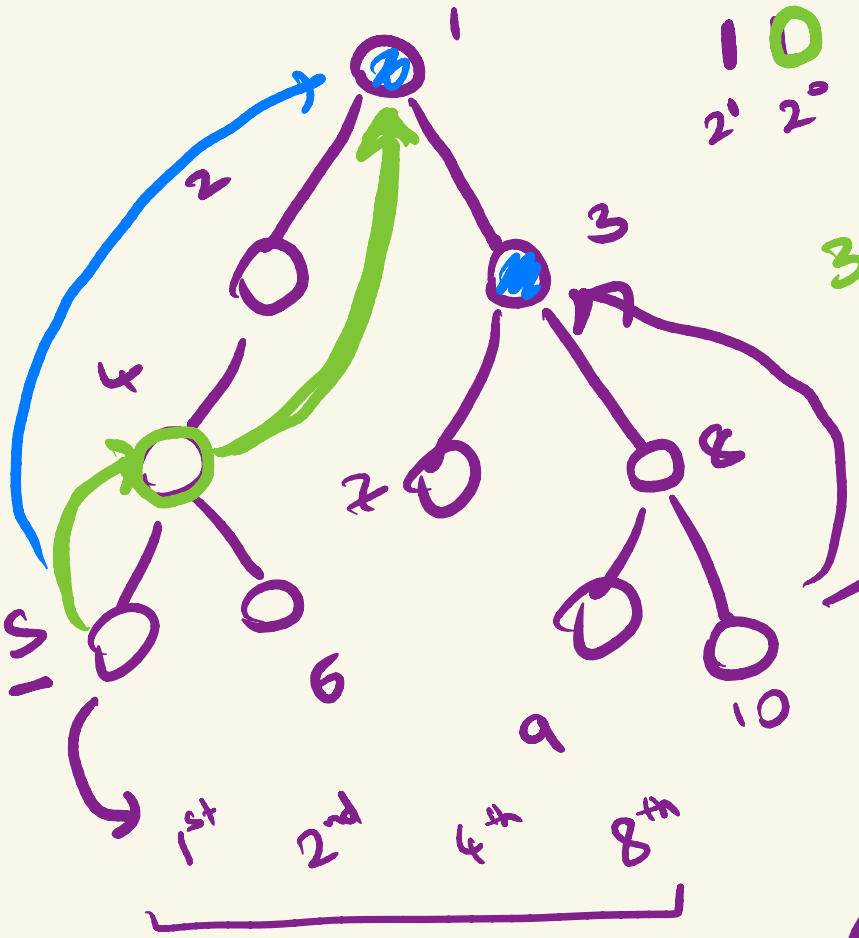
Implementation

```
int col[2] = {0, 0};

void dfs(int node, int parent, int color, vector<vector<int>>&adj) {
    col[color]++;
    for(auto &i : adj[node]) {
        if(i != parent) {
            dfs(i, node, color ^ 1, adj);
        }
    }
}

int main() {
    ios_base::sync_with_stdio(false);
    cin.tie(NULL);

    int n; cin >> n;

    vector<vector<int>>adj(n + 1);

    for(int i = 1; i < n; i++) {
        int u, v; cin >> u >> v;
        adj[u].push_back(v);
        adj[v].push_back(u);
    }

    dfs(1, 1, 0, adj);

    cout << 1ll * col[0] * col[1] - (n - 1) << "\n";
}
```

Tree

initial nodes





10

2^1 2^0

q queries

$3 = 2+1$ → x k^{th} ancestor

O(k) per query

10 → 4

10 / 2 → 3

5 / 3 → 1

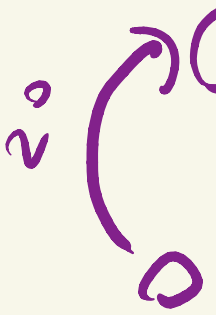
O(n·q) → TLE

$x \rightarrow \sum$ powers of 2

$O(\log_2 k)$

2^a parent

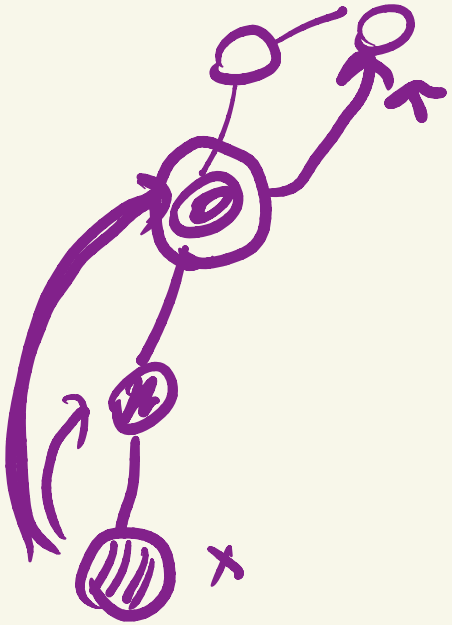
node $\rightarrow O(\log_2 n)$ parents



$17 \rightarrow 16 + 1$

$2^4 + 2^0$

$O(q \cdot \log_2(n))$



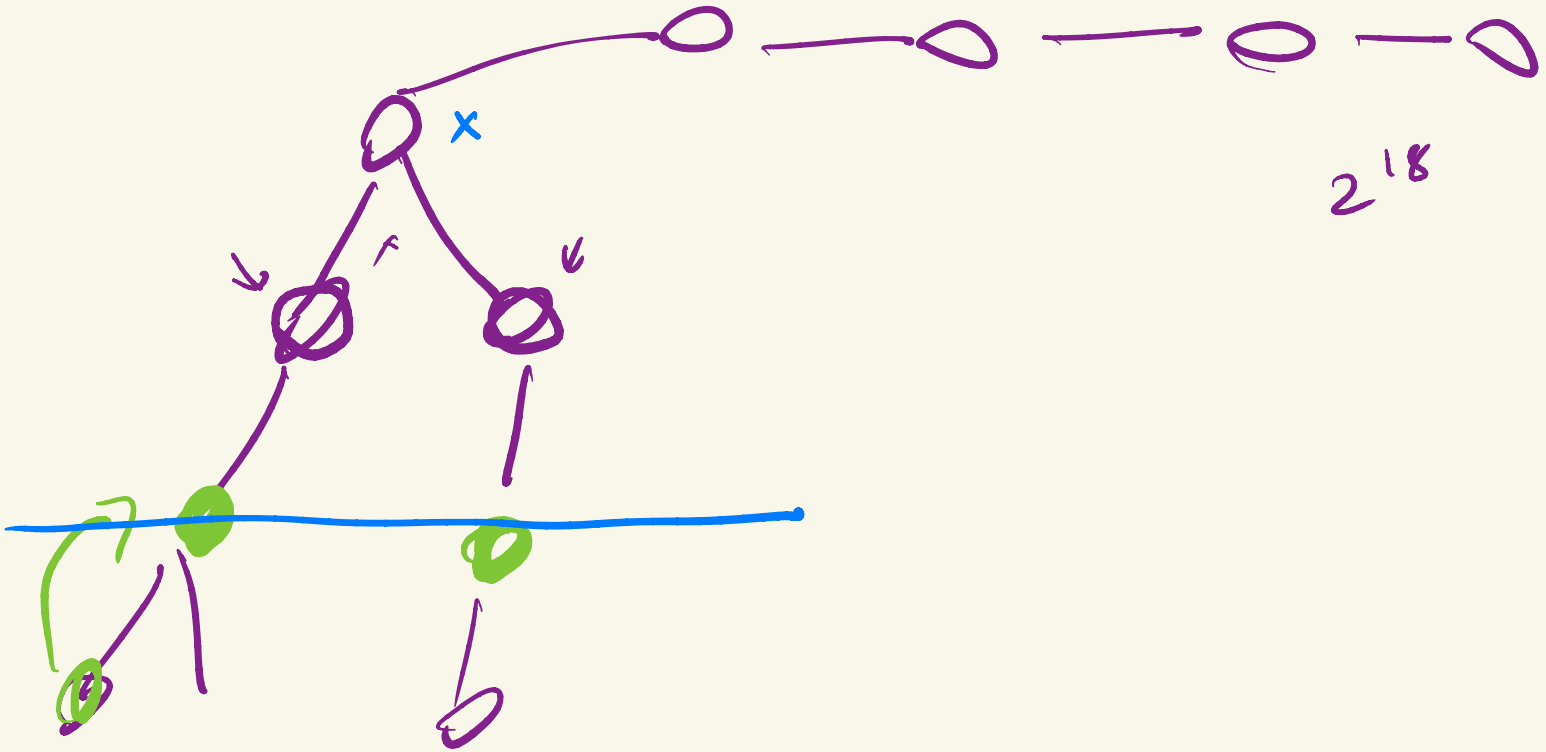
2^0 2^1 2^2

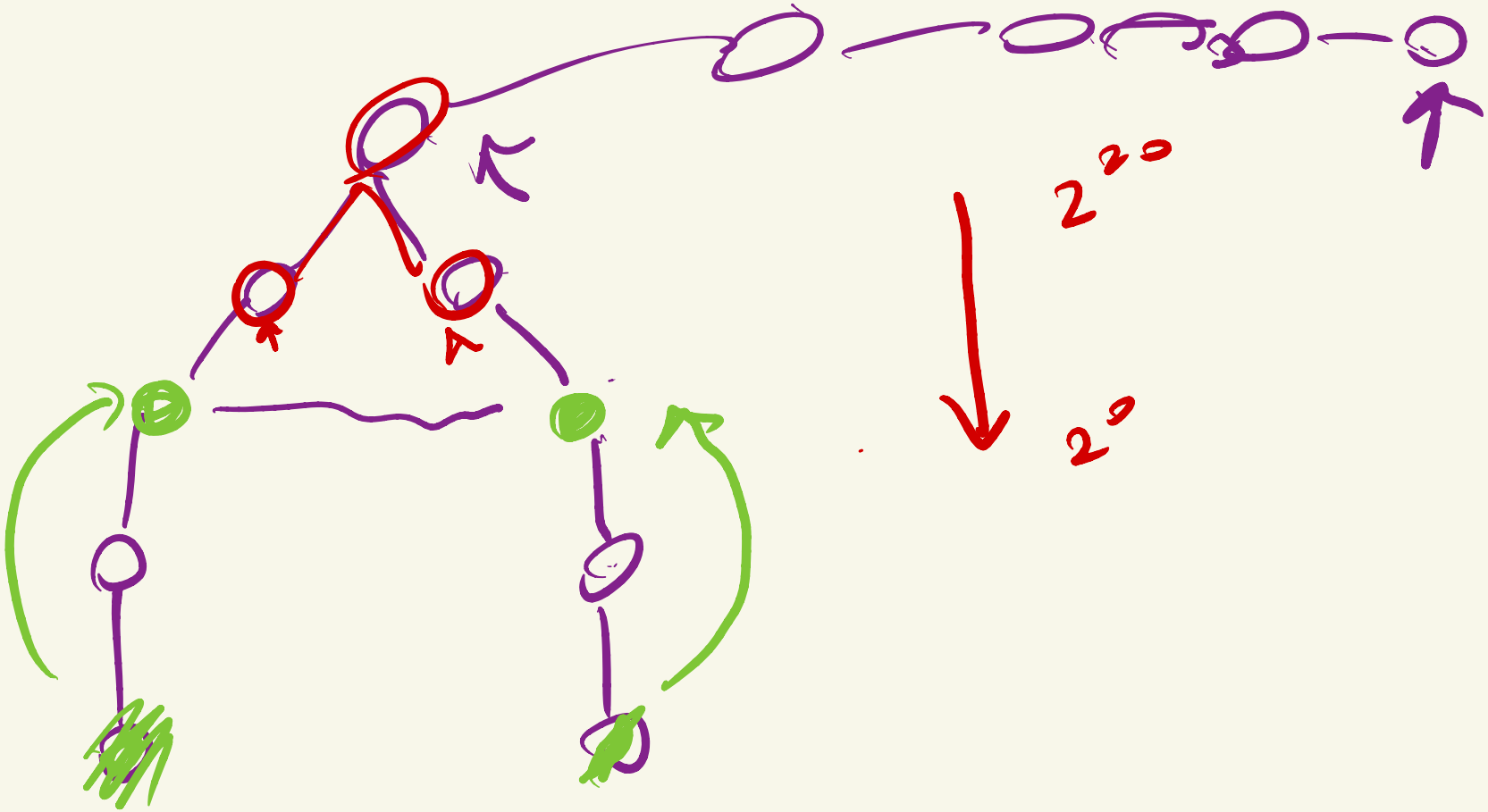
2^0 2^{0th} parent of your 2^{0th} parent

$2^i = 2^{i-1} + 2^{i-1}$

parent [x] [i] = parent [par[x] [i-1]] [i-1]

$O(n \log n + q \log n)$





8^{th} 7^{th} parent

$2^3 - 1 = 2^2 + 2^1 + 2^0$

$2^2 + 2^1 + 2^0 \Rightarrow 7$

Binary Lifting = Binary Exp