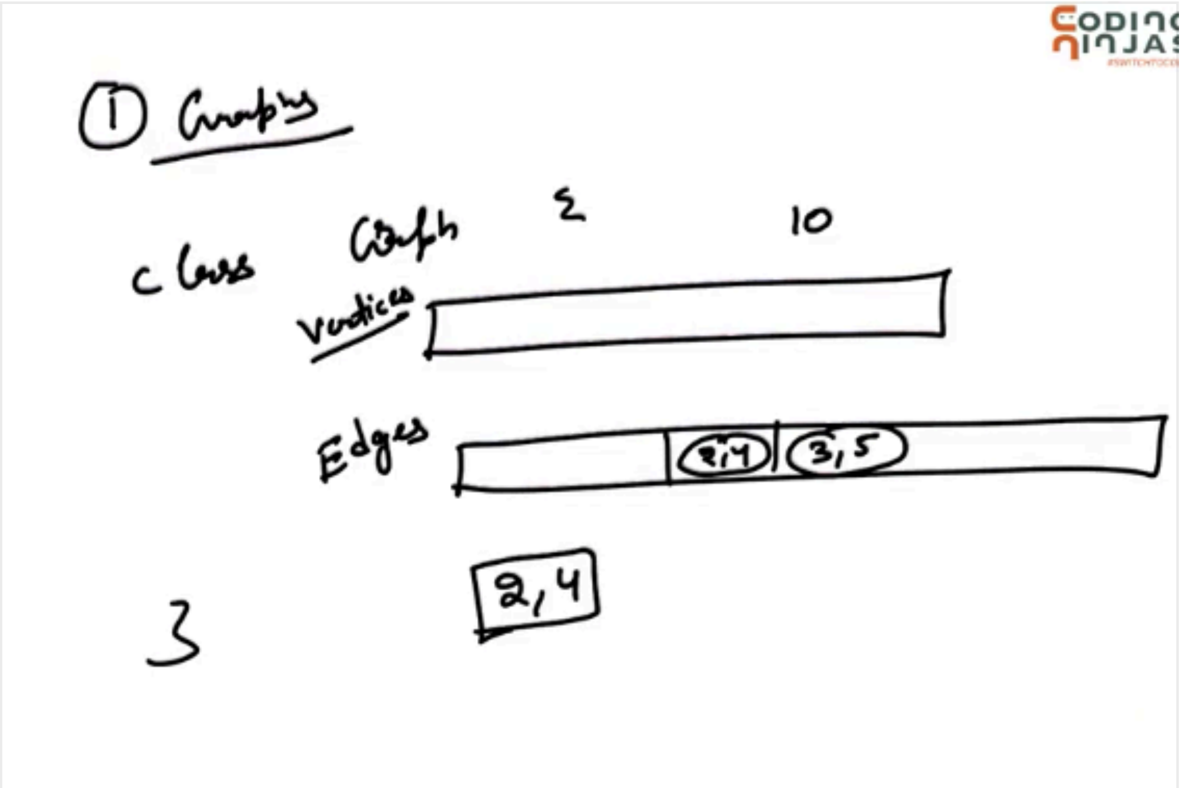
Graphs

Trees have hierarchal data , graphs give us freedom to store many different types of datas , a Tree is a connected graph without any cycle

- Degrees - number of edges going thru that vertex
- Connected Graph -> every two vertices have a path between them
- Connected Components - the set of vertices within a graph which are connected
- Tree-> a connected Graph which does not have a cycle !
- in a connected graph,with N vertices , the min edges we can have is N-1 and the max edges we can have is N*N, the latter is called complete graph (nC2)

Implementation

- Edge list and vertices list -O(n^2)



- Adjacency List
- Adjacency Matrix- not space efficient-huge waste of space in case of sparse graphs - but has ease of implementation
    - SPACE O(V*E)

DFS and BFS

```
int main() {
  cin >> n >> e;
  int** edges = new int*[n];
  for (int i = 0; i < n; i++) {
    edges[i] = new int[n];
    for (int j = 0; j < n; j++) {
      edges[i][j] = 0;
    }
  }

  for (int i = 0; i < e; i++) {
    int f, s;
    cin >> f >> s;
    edges[f][s] = 1;
    edges[s][f] = 1;
  }

  bool* visited = new bool[n];
  for (int i = 0; i < n; i++) {
    visited[i] = false;
  }
//  print(edges, n, 0, visited);

  printBFS(edges, n, 0);


  delete [] visited;
  for (int i = 0; i < n; i++) {
    delete [] edges[i];
  }
  delete [] edges;
}
```

```cpp
void print(int** edges, int n, int sv, bool* visited) {
  cout << sv << endl;
  visited[sv] = true;
  for (int i = 0; i < n; i++) {
    if (i == sv) {
      continue;
    }
    if (edges[sv][i] == 1) {
      if (visited[i]) {
        continue;
      }
      print(edges, n, i, visited);
    }
  }
}
```

```cpp
void printBFS(int** edges, int n, int sv) {
  queue<int> pendingVertices;
  bool * visited = new bool[n];
  for (int i = 0; i < n; i++) {
    visited[i] = false;
  }
  pendingVertices.push(sv);
  visited[sv] = true;
  while (!pendingVertices.empty()) {
    int currentVertex = pendingVertices.pop();
    cout << currentVertex << endl;
    for (int i = 0; i < n; i++) {
      if (edges[currentVertex][i] == 1 && !visited[i]) {
        pendingVertices.push(i);
        visited[i] = true;
      }
    }
  }
}
```

pendingVertices.front()

pendingVertices.pop()

```cpp
void DFS(int** edges, int n) {
  bool * visited = new bool[n];
  for (int i = 0; i < n; i++) {
    visited[i] = false;
  }
  for (int i = 0; i < n; i++) {
    if (!visited[i])
      printDFS(edges, n, i, visited);
  }
  delete [] visited;
}
```

```cpp
void BFS(int** edges, int n) {
  bool * visited = new bool[n];
  for (int i = 0; i < n; i++) {
    visited[i] = false;
  }
  for (int i = 0; i < n; i++) {
    if (!visited[i])
      printBFS(edges, n, i, visited);
  }
  delete [] visited;
}
```

How much complexity does hasPath have ? O(V+E) ?

GET PATH -BFS

for unconnected graph its actually shortest path !

    - we do bfs using queue and stop it as soon as end vertex is reached
    - to store parent we will actually keep a hashmap !

<u>Spanning Trees</u>

-> its a tree

- connected
- no cycles

Given an undirected and connected graph , and spanning tree is a tree with all the vertices, we can have multiple of these

- if we have n vertices in spanning tree we have n-1 edges
- if case this graph is weighted , the tree with minimum weights value is MST

<u>ADVANCED GRAPHS</u>

FINDING CONNECTED COMPONENETS

- We can do dfs/bfs and return a set of

sets

```
void dfs(vector<int>* edges, int start, unordered_set<int>* component, bool* visited) {
  visited[start] = true;
  component->insert(start);
  for (int i = 0; i < edges[start].size(); i++) {
    int next = edges[start][i];
    if (!visited[next]) {
      dfs(edges, next, component, visited);
    }
  }
}
```

dfs on adjacency list

```
unordered_set<unordered_set<int>*>* getComponents(vector<int>* edges, int n) {
  bool * visited = new bool[n];
  for (int i = 0; i < n; i++) {
    visited[i] = false;
  }
  unordered_set<unordered_set<int>*>* output = new unordered_set<unordered_set<int>*>(
  for (int i = 0; i < n; i++) {
    if (!visited[i]) {
      unordered_set<int>* component = new unordered_set<int>();
      dfs(edges, i, component, visited);
      output->insert(component);
    }
  }
  return output;
}
```

```
int main() {
  int n;
  cin >> n;
  vector<int>* edges = new vector<int>[n];
  int m;
  cin >> m;
  for (int i = 0; i < m; i++) {
    int j, k;
    cin >> j >> k;
    edges[j - 1].push_back(k - 1);
    edges[k - 1].push_back(j - 1);
  }
  unordered_set<unordered_set<int>*>* components = getComponents(edges, n);
  unordered_set<unordered_set<int>*>::iterator it1 = components->begin();
  while (it1 != components->end()) {
    unordered_set<int>* component = *it1;
    unordered_set<int>::iterator it2 = component->begin();
    while (it2 != component->end()) {
      cout << *it2 + 1 << " ";
      it2++;
    }
    cout << endl;
    delete component;
    it1++;
  }
  delete components;
}
```

CODING NINJAS

```cpp
bool hasPath(vector<vector<char>> &board, int n, int m) {
    bool foundPath = false;
    string word = "CODINGNINJA";
    vector<vector<bool>> used(n, vector<bool>(m, false));

    for (int i = 0; i < n; i++) {
        for (int j = 0; j < m; j++) {
            if (board[i][j] == word[0]) {
                foundPath = dfs(board, used, word, i, j, 1, n, m);
                if (foundPath) break;
            }
        }

        if (foundPath) break;
    }

    return foundPath;
}
```

```cpp
/*
    Time complexity: O(N * M)
    Space complexity: O(N * M)

    where N and M are the rows and columns respectively of the board
*/

int validPoint(int x, int y, int n, int m) {
    return (x >= 0 && x < n && y >= 0 && y < m);
}

bool dfs(vector<vector<char>> &board, vector<vector<bool>> &used, string &word, int x,
int y, int wordIndex, int n, int m) {
    if (wordIndex == 11) {
        return true;
    }

    used[x][y] = true;

    bool found = false;

    int dXdY[8][2] = {{-1,-1},{-1,0},{-1,1},{0,-1},{0,1},{1,-1},{1,0},{1,1}};

    for (int i = 0; i < 8; ++i) {
        int newX = x + dXdY[i][0];
        int newY = y + dXdY[i][1];

        if (validPoint(newX, newY, n, m) && board[newX][newY] == word[wordIndex] &&
!used[newX][newY]) {
            found = found | dfs(board, used, word, newX, newY, wordIndex + 1, n, m);
        }
    }

    used[x][y] = false;

    return found;
}
```

CONNECTING DOTS

```
#include<iostream>
using namespace std;
#include<vector>
// There exist 26 colours denoted by uppercase Latin characters (i.e. A,B,...,Z)
//find a cycle that contain dots of same colourcon
void dfs(vector<vector<char>> &board,
         vector<vector<bool>> &visited,
         int x,int y,
         int fromX,int fromY,
         char needColor,
         bool &foundCycle,
         int n,int m){
    if (x < 0 || x >= n || y < 0 || y >= m) {
        return;
    }
    if (board[x][y] != needColor) { return; }
    if (visited[x][y]) {
        foundCycle = true;
        return;
    }
    visited[x][y] = true;
    int dx[] = {1, -1, 0, 0};
    int dy[] = {0, 0, 1, -1};
    for (int i = 0; i < 4; ++i) {// RIGHT LEFT UP DOWN
        int nextX = x + dx[i];
        int nextY = y + dy[i];
        if (nextX == fromX && nextY == fromY) {continue;}// NOT GOING BACKWARDS IN THE CYCLE
        dfs(board, visited, nextX, nextY, x, y, needColor, foundCycle, n, m);
        // THE NEEDED COLOR STAYS THE SAME - X, Y BECOME NEW PARENT
    }
}
bool hasCycle(vector<vector<char>> &board, int n, int m) {
    bool foundCycle = false;
    vector<vector<bool>> visited(n, vector<bool>(m, false));
    for (int i = 0; i < n; i++) {
        for (int j = 0; j < m; j++) {
            if (!visited[i][j]) {
                dfs(board, visited, i, j, -1, -1, board[i][j], foundCycle, n, m);
                // this starting dfs has fromX and fromY as -1,-1, we are searching cycle for color board[i][j]
            }
        }
    }
    return foundCycle;
}
```