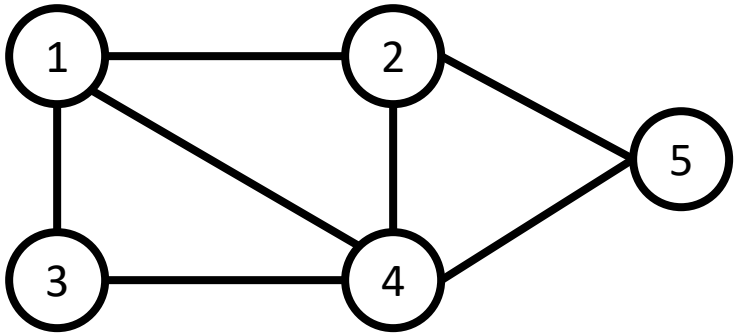
Graph algorithms

Basics of graphs

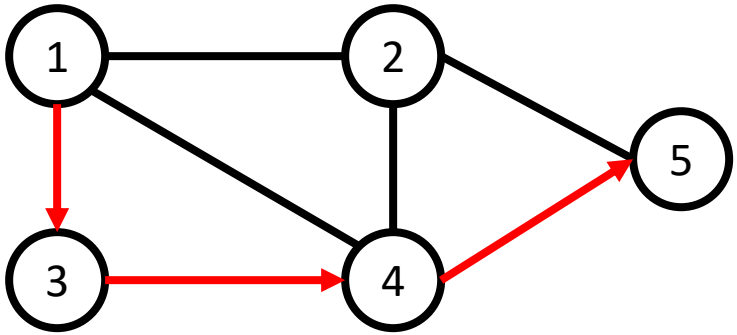
- หลายๆ ปัญหาโปรแกรมนั้นสามารถแก้ได้โดยการโมเดลปัญหานั้นเป็นปัญหากราฟ และใช้ graph algorithm ที่เหมาะสมในการแก้
- ตัวอย่างที่เห็นได้ง่ายของกราฟคือ network ของถนนและเมือง
- ในส่วนแรกจะทบทวนความรู้เบื้องต้นของกราฟก่อน

Graph Terminology

- Graph ประกอบด้วย โหนด และ เส้นเชื่อม
- ส่วนใหญ่เราจะใช้ตัวแปร n แทนจำนวนโหนดในกราฟและตัวแปร m แทนจำนวนเส้นเชื่อมในกราฟ เพื่อความง่ายเราก็จะกำหนดหมายเลขที่เป็นเลขจำนวนเต็มให้กับโหนด 1, 2, 3, ..., n หรือ 0 ถึง n–1
- ตัวอย่างเช่นกราฟต่อไปนี้มี 5 โหนด และมี 7 เส้นเชื่อม



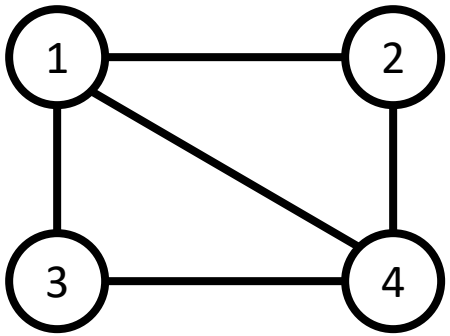
- **path** คือเส้นทางจากโหนด a ไปโหนด b ผ่านเส้นเชื่อมในกราฟ ความยาว(**length**) ของ path คือจำนวนของเส้นเชื่อมที่ใช้
- จากตัวอย่างก่อนหน้า path 1–>3–>4–>5 มีความยาว 3 จากโหนด 1 ไปโหนด 5



- path จะเป็น **cycle** ถ้าโหนดแรกและโหนดสุดท้ายเหมือนกัน ตัวอย่างเช่น cycle 1–>3–>4–>1 ส่วน path เป็น **simple** ถ้าแต่ละโหนดปรากฏไม่เกิน 1 ครั้งใน path

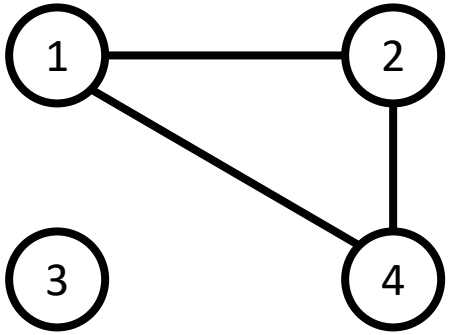
Connectivity

- กราฟเป็นกราฟเชื่อมต่อหรือ connected ถ้ามี path ระหว่างโหนดสองโหนดใดๆ นั่นคือไปถึงกันหมดหรือเชื่อมกันหมดนั่นเอง ตัวอย่างเช่น กราฟต่อไปนี้เป็นกราฟเชื่อมต่อ

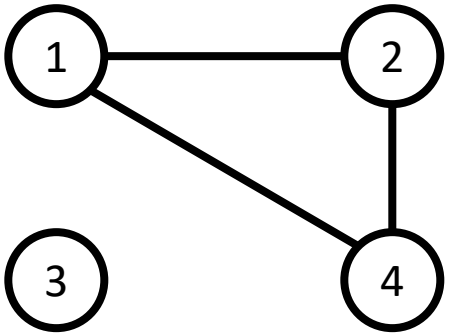


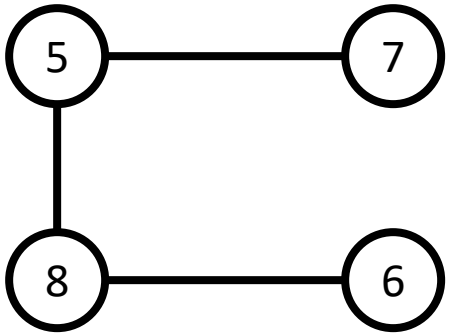
Connectivity

- ตัวอย่างต่อไปเป็นกราฟไม่เชื่อมต่อ เพราะว่ามันไม่มี path จากโหนด 3 ไปยังโหนดอื่น



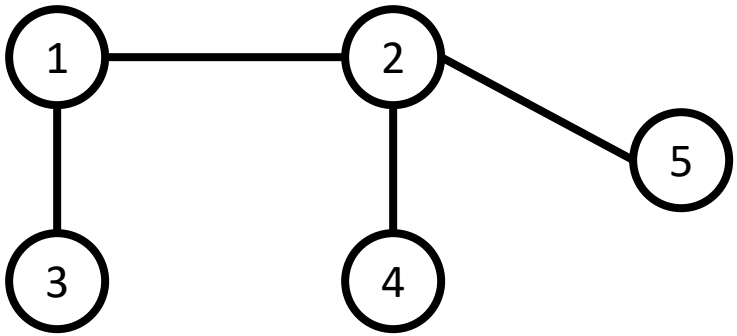
- ส่วนที่เชื่อมต่อกันของกราฟ เราจะเรียกว่า component ตัวอย่างต่อไปประกอบด้วย 3 components {1, 2, 4}, {3} {5, 6, 7, 8}





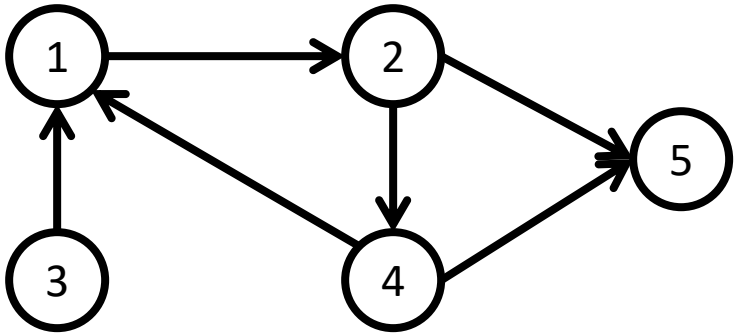
Tree

- Tree คือ connected graph ที่ประกอบด้วย n โหนดและ n–1 เส้นเชื่อม
- มี path ที่เป็น unique path ระหว่างสองโหนดใดๆ ใน tree ตัวอย่างกราฟต่อไปนี้เป็น tree



Edge directions

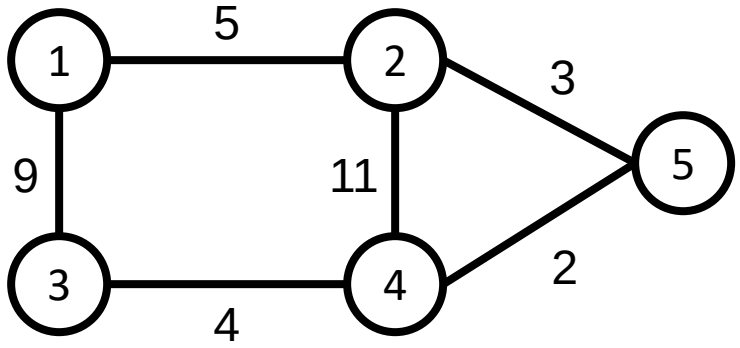
- กราฟจะเป็นกราฟแบบมีทิศทางหรือ directed graph ถ้าเส้นเชื่อมสามารถถูกสำรวจได้เพียงทิศทางเดียว ตัวอย่างกราแบบมีทิศทาง



- กราฟด้านบนมีเส้นทาง 3–>1–>2–>5 จากโหนด 3 ไปโหนด 5 แต่ไม่มีเส้นทางจากโหนด 5 ไปโหนด 3

Edge weights

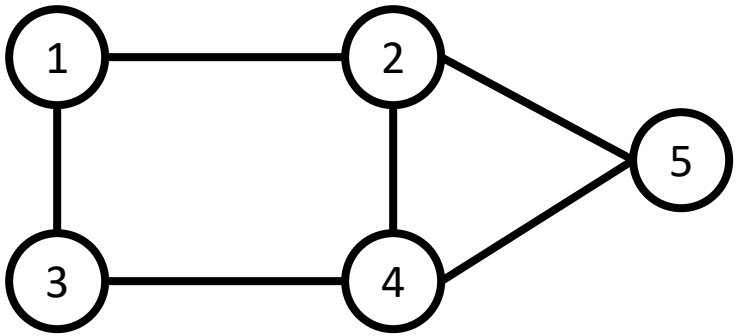
- ในกราฟแบบมีน้ำหนัก (weighted graph) เส้นเชื่อมจะถูกกำหนดค่าน้ำหนักไว้ (weight)
- weight อาจจะหมายถึงความยาวของเส้นเชื่อม
- ตัวอย่างกราฟแบบมีน้ำหนัก



- ความยาวของ path ใน weighted graph คือผลรวมของน้ำหนักบนเส้นเชื่อมบน path ตัวอย่างเช่นความยาวของ path 1–>3–>4–>5 คือ 15 ส่วน 1–>2–>5 คือ 8 ซึ่ง 1–>2–>5 เป็น shortest path

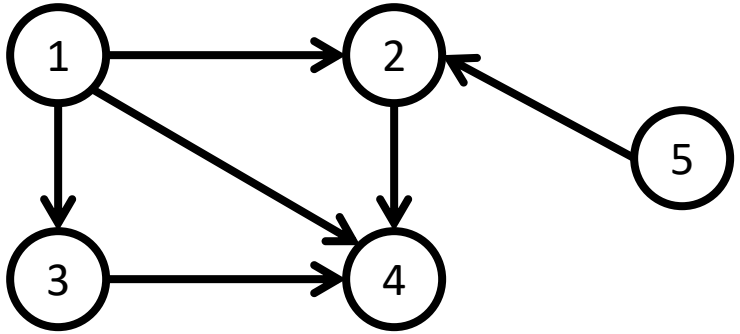
Neighbors and degrees

- โหนดสองโหนดเป็นโหนดเพื่อนบ้านกัน (neighbors) หรือติดกัน (adjacent) ถ้ามีเส้นเชื่อมระหว่างพวกมัน
- degree ของโหนดคือจำนวนของโหนดเพื่อนบ้าน ตัวอย่างเช่นกราฟต่อไปนี้โหนดเพื่อนบ้านของโหนด 2 คือ 1, 4 และ 5 ดังนั้นมันมี degree 3



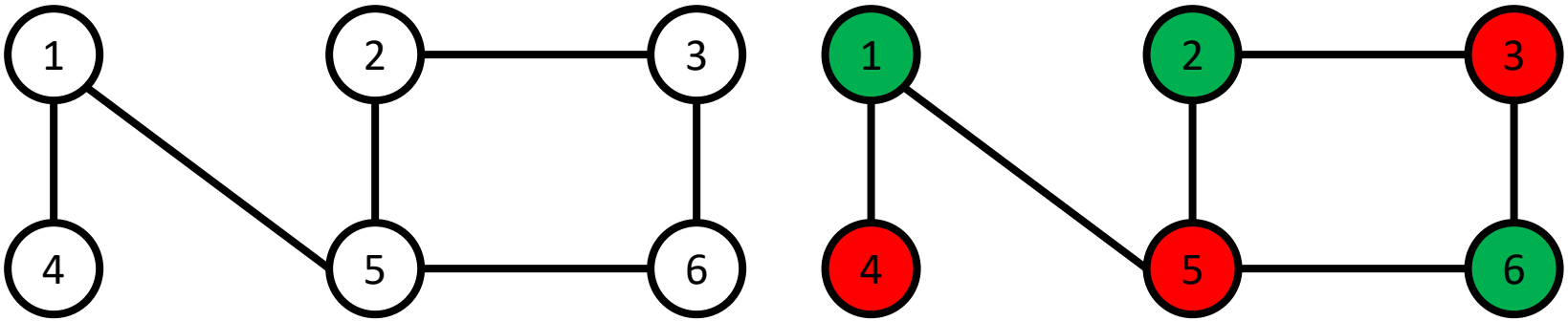
- ผลรวมของ degree ในกราฟจะเป็น 2m เสมอ เพราะว่าเส้นเชื่อมหนึ่งเส้นจะถูกนับสองครั้ง ดังนั้นผลรวม degree จึงเป็นคู่เสมออีกด้วย

- กราฟจะเป็น regular ถ้า degree ของทุกโหนดเป็นค่าคงที่ d
- กราฟจะเป็น complete graph ถ้าดีกรีของทุกโหนดเป็น n–1 นั่นคือกราฟเก็บทุกแบบของเส้นเชื่อมระหว่างโหนด
- ใน directed graph นั้น indegree ของโหนดคือจำนวนของเส้นเชื่อมที่มีจุดปลายที่โหนดนั้นและ outdegree ของโหนดคือจำนวนของเส้นเชื่อมที่มีจุดเริ่มต้นโหนดนั้น ตัวอย่างกราฟด้านล้าง indegree ของโหนด 2 เป็น 2 และ outdegree ของโหนด 2 เป็น 1

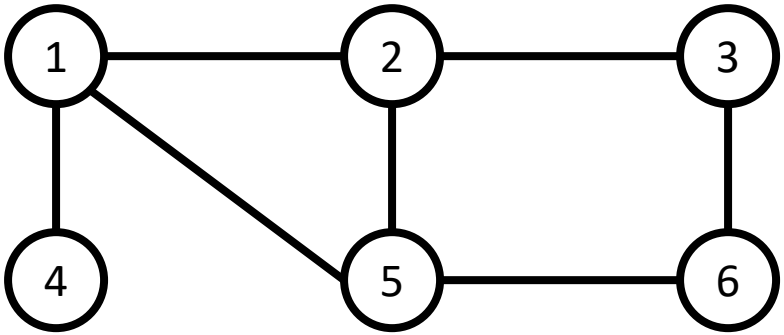


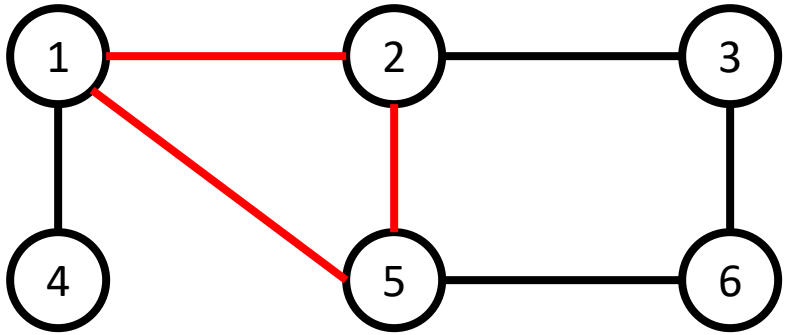
Coloring

- การระบายสีในกราฟ (coloring) แต่ละโหนดจะถูกกำหนดสีโดยที่โหนดเพื่อนบ้านจะต้องไม่เป็นสีเดียวกัน
- กราฟจะเป็นกราฟสองส่วน (bipartite) ถ้ามันสามารถระบายสีได้โดยใช้ 2 สี ข้อสังเกตถ้ากราฟเป็น bipartite เมื่อมันไม่มี cycle ที่มีจำนวนเส้นเชื่อมเป็นเลขคี่ ตัวอย่างเช่น



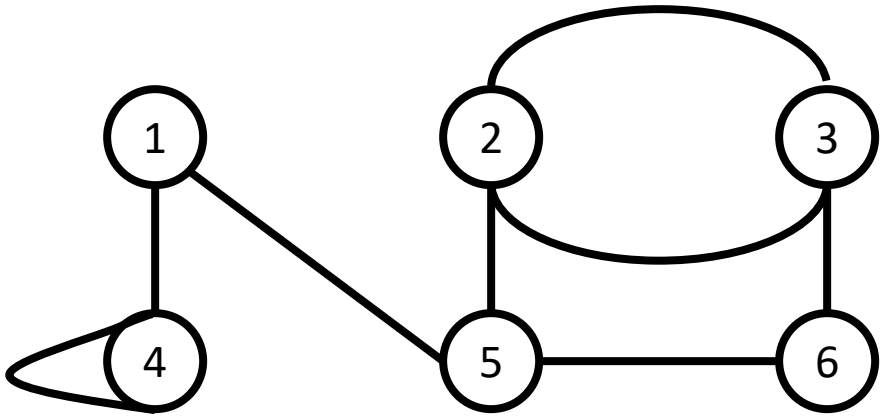
- อย่างไรก็ตามกราฟต่อไปนี้ไม่เป็น bipartite เพราะว่าไม่สามารถระบายสี cycle 3 โหนดโดยใช้สองสีได้





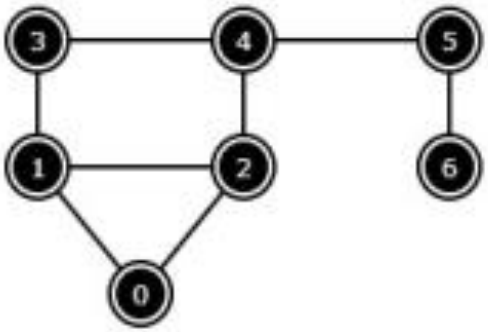
Simplicity

- กราฟจะเป็น simple graph ถ้าไม่มีเส้นเชื่อมที่จุดเริ่มต้นและจุดปลายเป็นจุดเดียวกัน และไม่มีเส้นเชื่อมมากกว่าหนึ่งเส้นที่เชื่อมระหว่างคู่ของโหนดใดๆ โดยทั่วไปแล้วเรามักจะสมมติว่ากราฟเป็น simple ตัวอย่างต่อไปแสดงกราที่ไม่ simple



Graph representation

- กราฟเป็นโครงสร้างข้อมูลที่ใช้กันแพร่หลาย ซึ่งถ้าเป็นใน TOI นั้น 6 ข้อ จะมีปัญหากราฟอย่างน้อย 2 ข้อ DP ก็อย่างน้อย 2 ข้อ
- กราฟ G=(V,E) ในรูปอย่างง่ายนั้น ประกอบด้วยเซตของโหนด (V) และเซตของเส้นเชื่อม (E ซึ่งเป็นการเก็บการเชื่อมต่อระหว่างโหนดใน V)



Adjancency Matrix

```
  0 1 2 3 4 5 6
0 0 2 5 0 0 0 0
1 2 0 7 1 0 0 0
2 5 7 0 0 4 0 0
3 0 1 0 0 3 0 0
4 0 0 4 3 0 9 0
5 0 0 0 0 9 0 8
6 0 0 0 0 0 8 0
```

Adjacency List

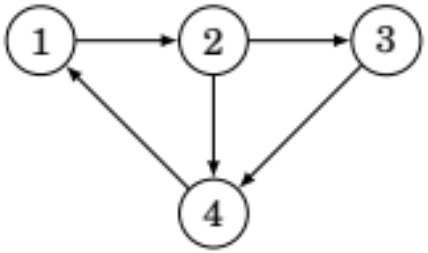
```
0: 1 2
1: 0 2 3
2: 0 1 4
3: 1 4
4: 2 3 5
5: 4 6
6: 5
```

Edge List

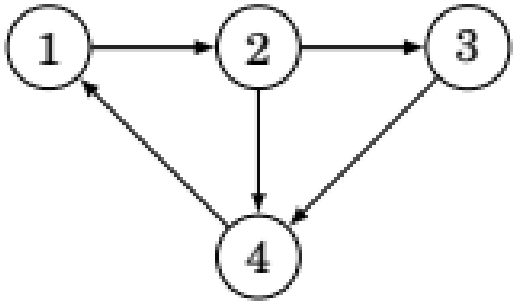
```
0:  0 1
1:  0 2
2:  1 0
3:  1 2
4:  1 3
5:  2 0
6:  2 1
7:  2 4
8:  3 1
9:  3 4
10: 4 2
11: 4 3
```

Adjacency list

- แต่ละโหนด x ในกราฟถูกกำหนด adjacency list ที่ประกอบด้วยโหนดที่มีเส้นเชื่อมจาก x
- adjacency list เป็นวิธีที่นิยมในการ represent กราฟที่มีประสิทธิภาพ
- วิธีที่เก็บที่สะดวกก็คือการประกาศ array of vectors
- vector<int> adj[N];
- ค่าคงที่ N เลือกโดยที่ทุก adjacency list สามารถเก็บค่าได้ ตัวอย่างเช่น



- เก็บอย่างไรดี



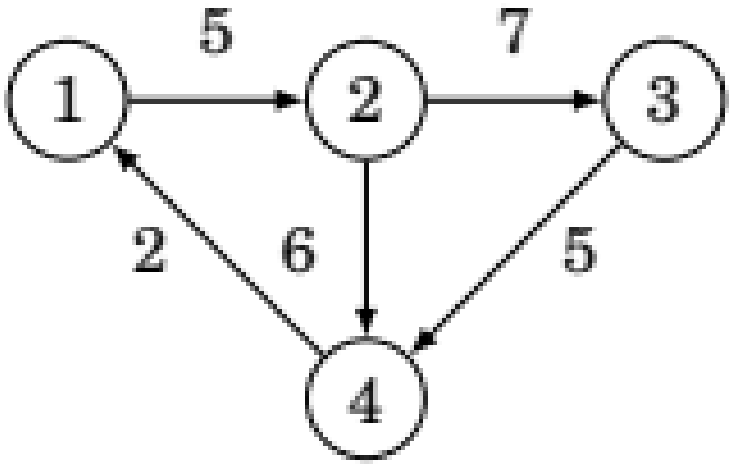

```
adj[1].push_back(2);
adj[2].push_back(3);
adj[2].push_back(4);
adj[3].push_back(4);
adj[4].push_back(1);
```

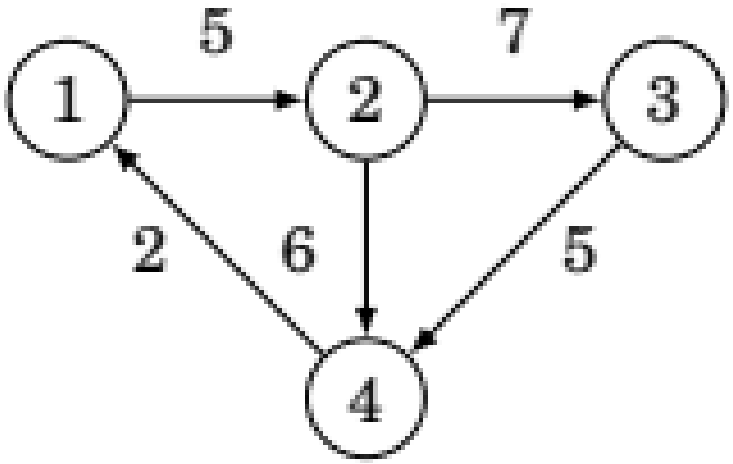
- ถ้ากราฟเป็น undirected graph เราก็เก็บเหมือนเดิมแต่เส้นเชื่อมต้องเก็บไปกลับ


```
adj[1].push_back(2);
adj[2].push_back(1);
```

ถ้าเป็น weighted graph เก็บไงดี

- สำหรับ weighted graph ก็เก็บด้วยวิธีนี้
- `vector <pair<int,int>> adj[N];`
- ในกรณีนี้ adjacency list ของโหนด a เก็บ pair(b,w) เมื่อมีเส้นเชื่อมจากโหนด a ไปโหนด b ด้วย weight w
- พิจารณาตัวอย่างต่อไปนี้






```
adj[1].push_back(2,5);
adj[2].push_back(3,7);
adj[2].push_back(4,6);
adj[3].push_back(4,5);
adj[4].push_back(1,2);
```

- ข้อดีของการใช้ adjacency list คือเราสามารถหาโหนดที่เราจะย้ายไปยังโหนดต่อไปผ่านทางเส้นเชื่อมได้อย่างมีประสิทธิภาพ ตัวอย่างเช่นเราสามารถเขยน loop ที่ไปยังทุกโหนดจากโหนดได้ดังนี้

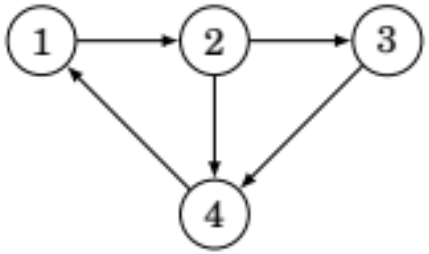
```
for(auto u:adj[s]){
    //process node u
}
```

Adjacency matrix

- Adjacency matrix เป็น array 2 มิติที่บ่งบอกว่าเส้นเชื่อมในกราฟมีเส้นใดบ้าง เราสามารถตรวจสอบจาก adjacency matrix ได้อย่างมีประสิทธิภาพถ้ามีเส้นเชื่อมระหว่างโหนดสองโหนดใดๆ
- matex สามารถเก็บได้ง่ายๆ โดยใช้ array

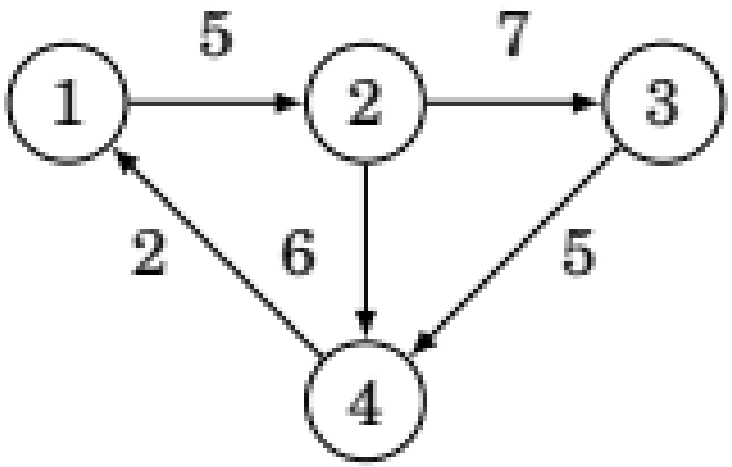
```
int adj[N][N];
```

- โดยที่แต่ละค่าของ adj[a][b] บ่งบอกว่ากราฟมีเส้นเชื่อมจาก a ไป b ถ้าเส้นเชื่อมนั้นมีในกราฟแล้ว adj[a][b] = 1 แต่ถ้าไม่มี adj[a][b] = 0
- ตัวอย่างเดิม เก็บอย่างไรดี



	1	2	3	4
1	0	1	0	0
2	0	0	1	1
3	0	0	0	1
4	1	0	0	0

- ถ้ากราฟเป็น weighted graph แล้ว adjacency matrix ก็เก็บได้โดยเก็บค่า weight ของเส้นเชื่อมถ้ามีเส้นเชื่อมนั้น ตัวอย่างเช่น

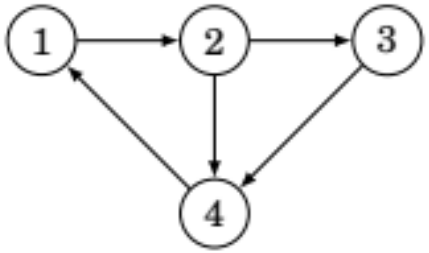


	1	2	3	4
1	0	5	0	0
2	0	0	7	6
3	0	0	0	5
4	2	0	0	0

- ข้อเสียของ adjacency matrix คือว่า matrix นั้นต้องเก็บข้อมูล n^2 ตัว และโดยส่วนมากเป็น 0

Edge list

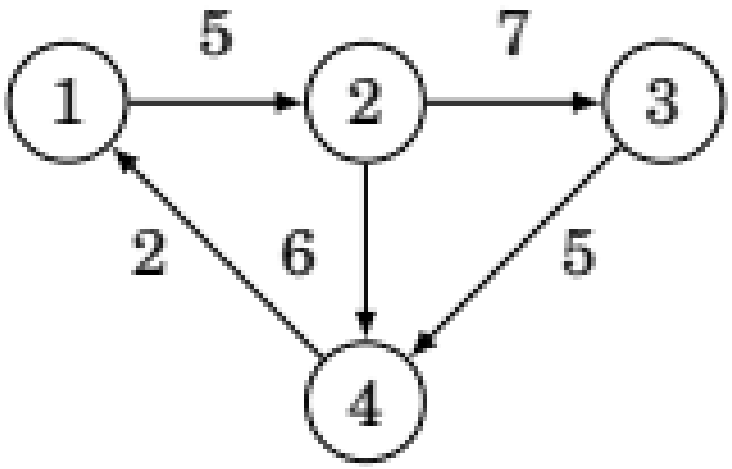
- Edge list นั้นเก็บทุกเส้นเชื่อมของกราฟในลำดับบางลำดับ
- นี่เป็นวิธีการเก็บที่สะดวกมากในการแทนกราฟถ้า algorithm ประมวลผลทุกเส้นเชื่อมของกราฟและมันไม่จำเป็นต้องหาเส้นเชื่อมที่เริ่มต้นของโหนดที่กำหนดมาให้
- `vector<pair<int,int>> edges;`
- เมื่อแต่ละ pair(a,b) แทนว่ามีเส้นเชื่อมจากโหนด a ไป โหนด b




```
edges.push_back({1,2});
edges.push_back({2,3});
edges.push_back({2,4});
edges.push_back({3,4});
edges.push_back({4,1});
```

ถ้าเป็น weighted graph ก็ปรับนิดหน่อย

- `vector<tuple<int,int,int>> edges;`
- แต่ละสมาชิกใน list อยู่ในรูปขของ (a, b, w) ความหมายคือ มีเส้นเชื่อมจากโหนด a ไปโหนด b ด้วยน้ำหนัก ตัวอย่างเช่น




```
edges.push_back({1,2,5});
edges.push_back({2,3,7});
edges.push_back({2,4,6});
edges.push_back({3,4,5});
edges.push_back({4,1,2});
```

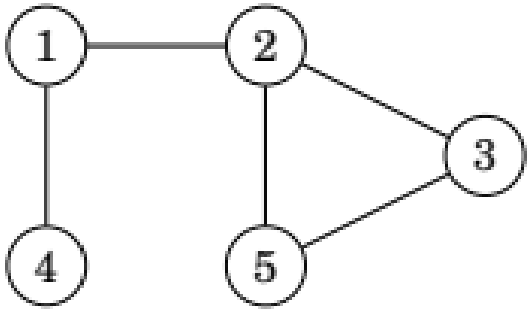
Graph Traversal

- ในหัวข้อนี้เราจะมาทบทวน graph algorithms พื้นฐานสองอันได้แก่ Depth-first search แล้ Breadth-first search
- ทั้งสอง algorithms นั้นจะกำหนดโหนดเริ่มต้นในกราฟมาให้และทั้งคู่จะไปแวะทุกโหนดที่สามารถไปถึงได้จากโหนดเริ่มต้น
- ความแตกต่างของสอง algorithms นี้คือลำดับในการแวะโหนด

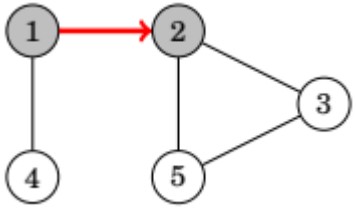
Depth–first search

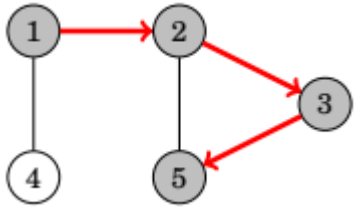
- Depth–first search (DFS) เป็นอัลกอริทึมที่ง่ายอย่างหนึ่งในการท่องไปในกราฟ traversing a graph
- อัลกอริทึมนี้เริ่มต้นที่โหนดเริ่มต้นโหดนหนึ่ง จากนั้น DFS จะท่องไปในกราฟในเชิงลึกก่อน(ไปให้สุดก่อน) โดยทุกครั้งที่ DFS ไปพบจุดทางแยก (Brancing point) หรือจุดที่มีเพื่อนบ้านมากกว่าหนึ่ง DFS จะเลือกหนึ่งในเพื่อนบ้านที่ยังไม่เคยไป และไปเยี่ยมเพื่อนบ้านนั้น
- DFS จะทำซ้ำกระบวนการนี้และไปต่อลึกเรื่อยๆ จนกระทั่งมันไปถึงโหนดที่ไม่สามารถไปต่อได้อีก เมื่อเกิดเหตุการณ์นี้ขึ้น DFS จะย้อนกลับ (Backtrack) แล้วค้นหาเพื่อนบ้านที่ยังไม่เคยแวะต่อ

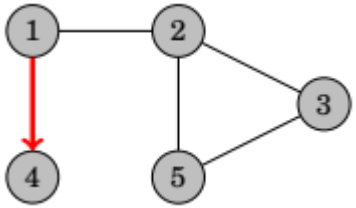
- ตัวอย่างของ DFS



เราอาจจะเริ่มต้นที่โหนดใดก็ได้ในกราฟ สำหรับตัวอย่างนี้เราจะเริ่มที่ 1







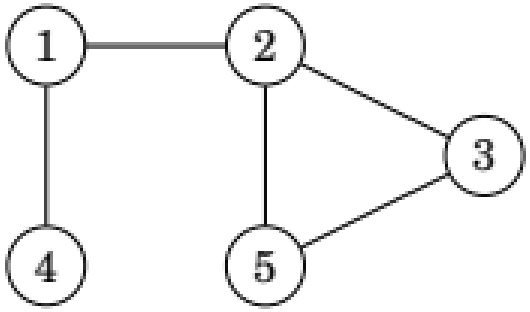
- จากรูปเพื่อนบ้านของ 5 คือ 2 และ 3 แต่การค้นหาไปพบสองโหนดนั้นแล้ว ทำให้ถึงเวลาที่ต้องย้อนกลับมายังโหนดก่อนหน้าซึ่งคือโหนด 3 เพื่อนบ้านของโหนด 3 ก็สำรวจหมดแล้วก็ย้อนขึ้นไปอีก โหนดสองก็สำรวจหมดแล้ว ก็ย้อนไปอีก โหนด 1 ยังสำรวจไม่หมดจึงไปโหนด 4

- หลังจากนั้นการค้นหาก็หยุดลงเมื่อสำรวจครบทุกโหนด
- เวลาในการทำงานของ DFS คือ O(n+m) เพราะว่าอัลกอริทึมนั้นประมวลผลทุกโหนดและทุกเส้นเชื่อมเพียงครั้งเดียว โดยการใช้งาน dfs_num ในการเช็ค

- การ implement DFS สามารถทำได้อย่างง่ายด้วย recursion
- ซึ่งฟังก์ชัน dfs นั้นเริ่มต้นจากโหนดที่กำหนดให้
- สมมติว่ากราฟเก็บแบบ adjacency list ใน array vector<int> adj[N];
- และเก็บ array สำรวจ vector <int> dfs_num[N]; โดยเราจะให้ dfs_num เป็นตัวแปร global ที่เอาไว้แยกสถานะของแต่ละโหนด โดยให้ –1 แทน unvisited และ ให้ 1 แทน visited
- เริ่มต้นเราจะให้ dfs_num ทุกค่าเป็น unvisited

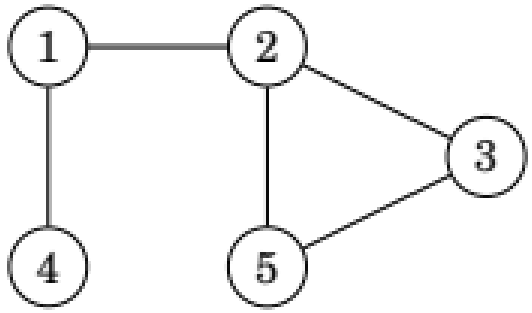
DFS(แบบที่ 1)

```
const int N = 100010;
vector<int> dfs_num;
vector<int> adj[N];
void dfs(int s){
    printf("%d\n", s);
    dfs_num[s] = true;
    for (auto u : adj[s]) {
        if(dfs_num[u]==-1)
        dfs(u);
    }
}
```



DFS(แบบที่ 1 ต่อ)

```
int main(){
    int n, m; // nodes, edges
    scanf("%d%d", &n, &m);
    dfs_num.assign(n, -1);
    for (int i = 0; i < m; ++i) {
        int u, v;
        scanf("%d%d", &u, &v);
        adj[u].push_back(v);
        adj[v].push_back(u);
    }
    int s; // start
    scanf("%d", &s);
    dfs(s);
    return 0;
}
```



https://bit.ly/2XumNHn

DFS(แบบที่ 2 Adjacency list แบบ weight)

```
#include <bits/stdc++.h>
using namespace std;
typedef pair<int, int> ii;
typedef vector<ii> vii;
typedef vector<int> vi;
#define DFS_WHITE -1
#define DFS_BLACK 1
vector<vii> AdjList;
vi dfs_num;     // this variable has to be global, we cannot put it in recursion
void dfs(int u) {
  printf(" %d", u);                                    // this vertex is visited
  dfs_num[u] = DFS_BLACK;      // important step: we mark this vertex as visited
  for (int j = 0; j < (int)AdjList[u].size(); j++) {
    ii v = AdjList[u][j];                          // v is a (neighbor, weight) pair
    if (dfs_num[v.first] == DFS_WHITE)             // important check to avoid cycle
      dfs(v.first);      // recursively visits unvisited neighbors v of vertex u
  }
}
```

DFS(แบบที่ 2 ต่อ)

```
int main()
{
  int V, total_neighbors, id, weight;
  scanf("%d", &V);
  AdjList.assign(V, vii());
// assign blank vectors of pair<int, int>s to AdjList
  for (int i = 0; i < V; i++) {
    scanf("%d", &total_neighbors);
    for (int j = 0; j < total_neighbors; j++) {
      scanf("%d %d", &id, &weight);
      AdjList[i].push_back(ii(id, weight));
    }
  }
```

```
  dfs_num.assign(V, DFS_WHITE);
// this sets all vertices' state to DFS_WHITE
  for (int i = 0; i < V; i++)
// for each vertex i in [0..V-1]
    if (dfs_num[i] == DFS_WHITE){
// if that vertex is not visited yet
      dfs(i);
      printf("\n");
    }
  return 0;
}
```

https://gist.github.com/cjakarin/ef7d2dc6094225ea51d003adec4ea44d

https://bit.ly/2GV6G0j

5

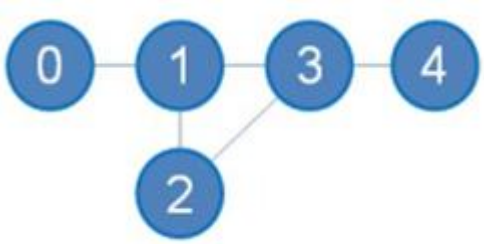
1 1 0

3 0 0 2 0 3 0

2 1 0 3 0

3 1 0 2 0 4 0

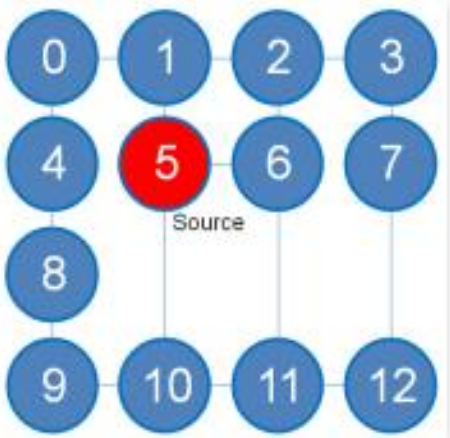
1 3 0

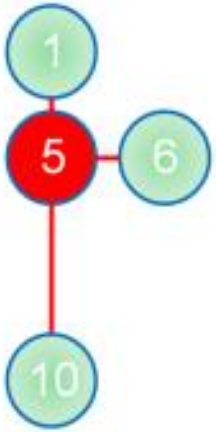


Breadth–first search

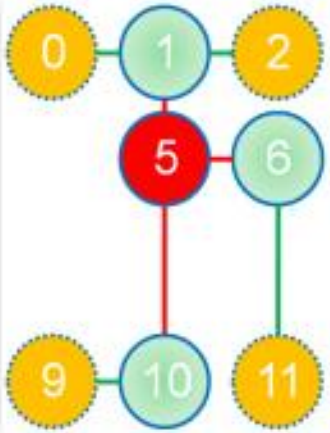
- Breadth–first search (BFS) เป็นอัลกอริทึมอีกอันในการท่องไปในกราฟ
- โดยเริ่มต้นจากโหนดเริ่มต้น(source node) BFS จะท่องไปในกราฟแบบแผ่กว้างก่อน นั่นคือ BFS จะไปเยี่ยมโหนดที่เป็นเพื่อนบ้านโดยตรงของโหนดเริ่มต้นก่อน (เป็นชั้นแรก) จากนั้นจะไปเยี่ยมเพื่อนบ้านของเพื่อนบ้านโดยตรง(เป็นชั้นที่สอง) ทำเช่นนี้ต่อไปเรื่อยๆ ทีละชั้น
- BFS เริ่มต้นด้วยการเพิ่ม souce node ลงไปใน queue จากนั้นจะทำงานกับ queue ดังนี้
  - หยิบโหนดหน้าสุด u ออกมาจาก queue
  - enqueue โหนดเพื่อนบ้านของ u ที่ยังไม่ได้ visit
  - mark ว่า visit

- จากการใช้ queue BFS นั้นจะไปเยี่ยมโหนด s จากนั้นไปยังทุกโหนดในกลุ่มที่เชื่อมต่อกัน (connected component) ที่มี s อยู่ ทีละชั้น
- BFS ทำงานใน O(V+E) บนกราฟที่ใช้ Adjacency List และ $O(V^2)$ บนกราฟที่ใช้ Adjacency Matrix
- ในการ implemet BFS นั้นไม่ยาก ถ้าเราใช้ C++ STL โดยเรา ใช้ queue เพื่อเก็บลำดับโหนดที่ไปเยี่ยมและ vector<int> ในการบันทึก ว่าโหนดนั้นถูก visit ไปแล้วหรือยัง ซึ่งในเวลาเดียวกันนั้นเราสามารถบันทึกระยะทาง(ชั้นที่) ของแต่ละโหนดเริ่มจาก source ได้

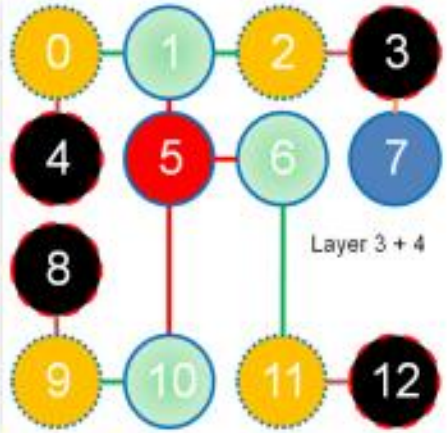




Layer 1



Layer 2



Layer 3 + 4

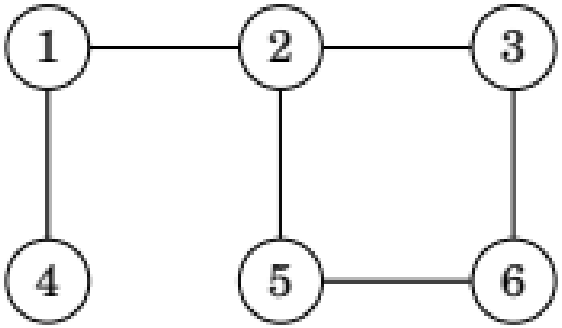
- 13 16
- 0 1 1 2 2 3 0 4 1 5 2 6 3 7 5 6
- 4 8 8 9 5 10 6 11 7 12 9 10 10 11 11 12

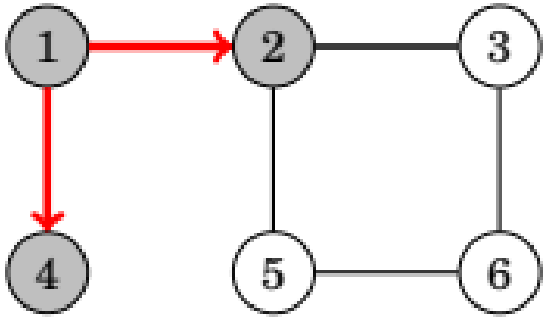
```
typedef pair<int, int> ii;
typedef vector<ii> vii;
typedef vector<int> vi;
int V, E, a, b, s;
vector<vii> AdjList;
vi p;                          // addition: the predecessor/parent vector
int main() {
  scanf("%d %d", &V, &E);
  AdjList.assign(V, vii());
// assign blank vectors of pair<int, int>s to AdjList
  for (int i = 0; i < E; i++) {
    scanf("%d %d", &a, &b);
    AdjList[a].push_back(ii(b, 0));
    AdjList[b].push_back(ii(a, 0));
  }
  // we start from this source
  s = 5;
```

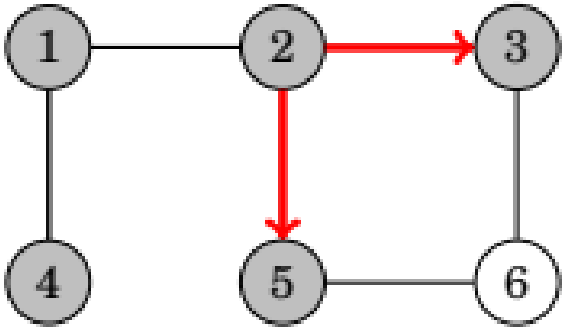
```
  vi dist(V, 1000000000); dist[s] = 0;   // distance to source is 0
  queue<int> q; q.push(s);                // start from source
  p.assign(V, -1); // to store parent information
  int layer = -1; // for our output printing purpose
  while (!q.empty()) {
    int u = q.front(); q.pop();                  // queue: layer by layer!
    if (dist[u] != layer) printf("\nLayer %d: ", dist[u]);
    layer = dist[u];
    printf("visit %d, ", u);
    for (int j = 0; j < (int)AdjList[u].size(); j++) {
      ii v = AdjList[u][j];                  // for each neighbors of u
      if (dist[v.first] == 1000000000) {
        dist[v.first] = dist[u] + 1;         // v unvisited + reachable
        p[v.first] = u;   //addition: the parent of vertex v->first is u
        q.push(v.first);                     // enqueue v for next step
   }
  }
 }
 return 0;
}
```

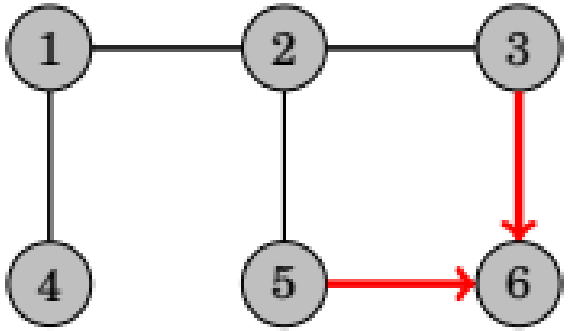
https://bit.ly/2GTkeJB

- ตัวอย่างการทำงานสมมติว่าเริ่มที่โหนด 1









- เราคำนวณระยะทางจากโหนดเริ่มต้นไปยังทุกโหนดในกราฟได้ ซึ่งระยะทางที่ได้เป็น

node	distance
1	0
2	1
3	2
4	1
5	2
6	3

- เราสมมติว่าเก็บกราฟเป็น adjacency list และเก็บข้อมูลอื่นๆ ดังนี้
- queue<int> q;
- bool visited[N];
- int distance[N];

- Queue q นั้นเก็บโหนดที่จะถูกประมวลผลในลำดับที่เพิ่มขึ้นจากระยะทาง
- โหนดใหม่จะถูกเพิ่มที่ท้าย queue และโหนดที่หัว queue จะคือโหนดถัดไปที่จะถูกประมวลผล
- array visited เป็นตัวบ่งบอกว่าโหนดใดถูก search แล้ว
- array distance เก็บระยะทางจาก start node ไปยังทุกโหนดในกราฟ
- code ต่อไปสมมติว่าเริ่มต้นที่โหนด x

BFS(1)

```
const int N = 100010;
bool visited[N];
vector<int> adj[N];
int dis[N]={0};
int main()
{
    int n, m; // nodes, edges
    scanf("%d%d", &n, &m);
    for (int i = 0; i < m; ++i) {
        // assuming 1-based index
        int u, v;
        scanf("%d%d", &u, &v);
        adj[u].push_back(v);
        adj[v].push_back(u);
    }
```

BFS(2)

```
    int s; // start
    scanf("%d", &s);
    queue<int> togo;
    togo.push(s);
    dis[s]=0;
    while (!togo.empty()) {
        int u = togo.front();
        togo.pop();
        if (visited[u])
            continue;
        visited[u] = true;
        printf("%d\n", u);
        for (auto v : adj[u]) {
            if (!visited[v]){
                togo.push(v);
                  dis[v]=dis[u]+1;
              }
        }
    }
    return 0;
}
```

https://bit.ly/2Tq4Eeh