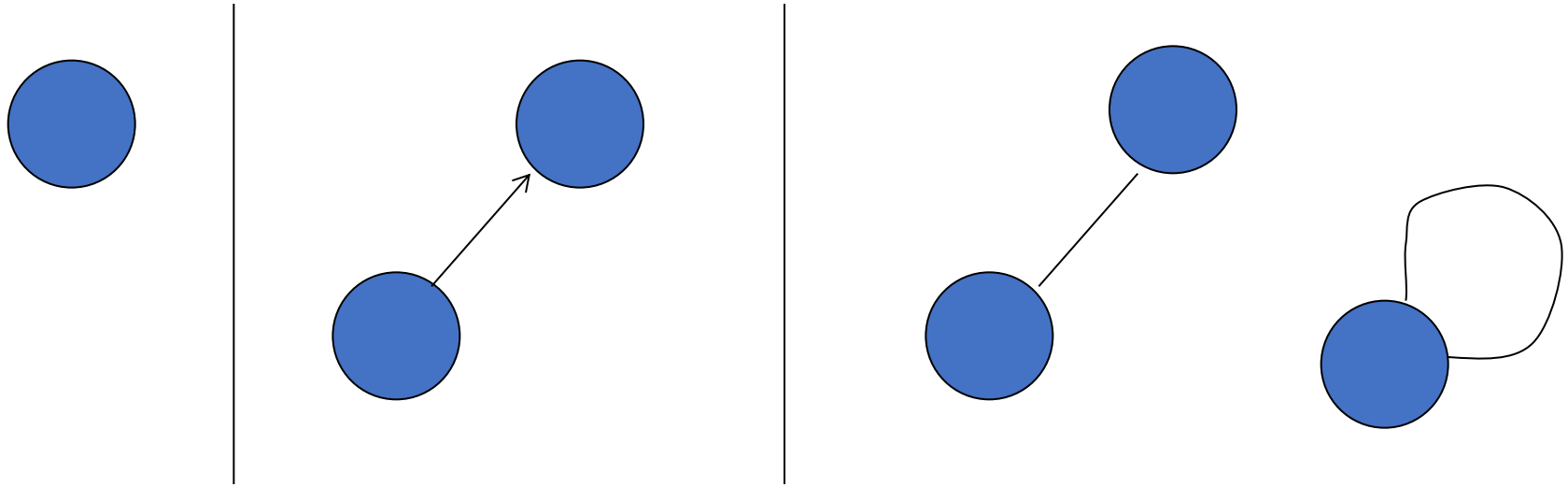
Graph

IIITS

Graphs

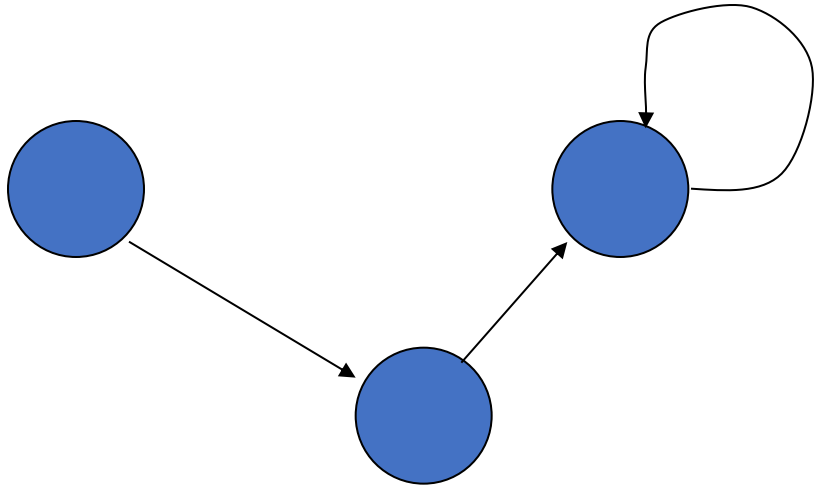
- Tuple <V,E>
- V is set of vertices
- E is a binary relation on V
  - Each edge is a tuple < v1,v2 >, where v1,v2 in V
- $|E| =< |V|^2$



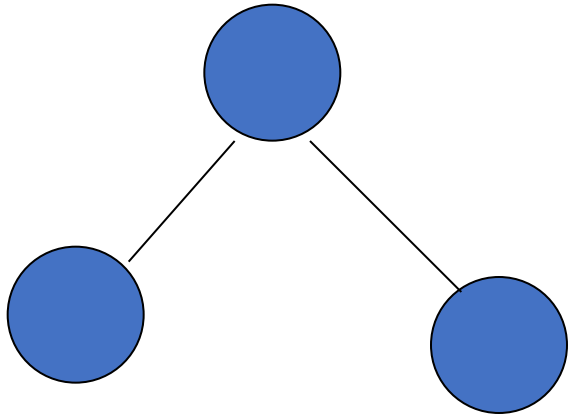
Directed and Undirected Graphs

- Directed graph:
  - < v1, v2 > in E is ordered, i.e., a relation (v1,v2)
- Undirected graph:
  - < v1, v2 > in E is un-ordered, i.e., a set { v1, v2 }
- Directed + Undirected

Examples of graphs



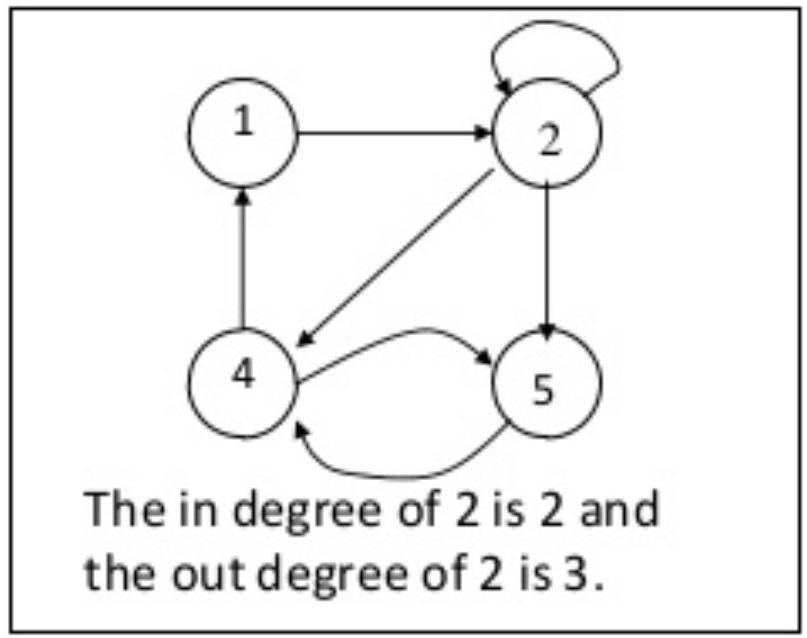
Directed



Undirected

Degree (Directed Graphs)

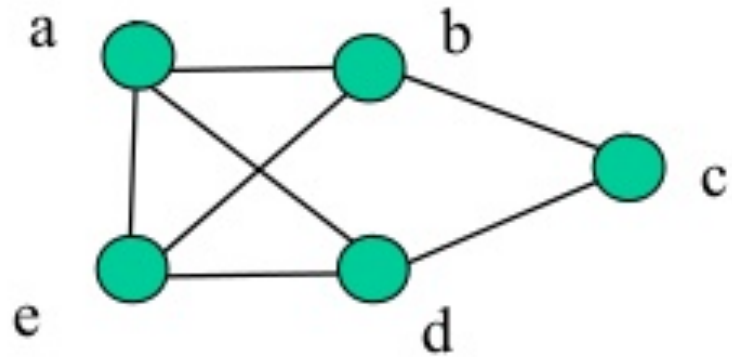
- In degree: Number of edges entering a node
- Out degree: Number of edges leaving a node
- Degree = Indegree + Outdegree



The in degree of 2 is 2 and
the out degree of 2 is 3.

Path and Cycle

- ***Path*** : a sequence of distinct vertices such that two consecutive vertices are adjacent
  - Example: (a, d, c, b, e) is a path
  - (a, b, e, d, c, b, e, d) is not a path; it is a walk
- ***Cycle*** : a closed Path
  - Example: (a, d, c, b, e, a) is a cycle



Connected Graph / Graph Connectivity

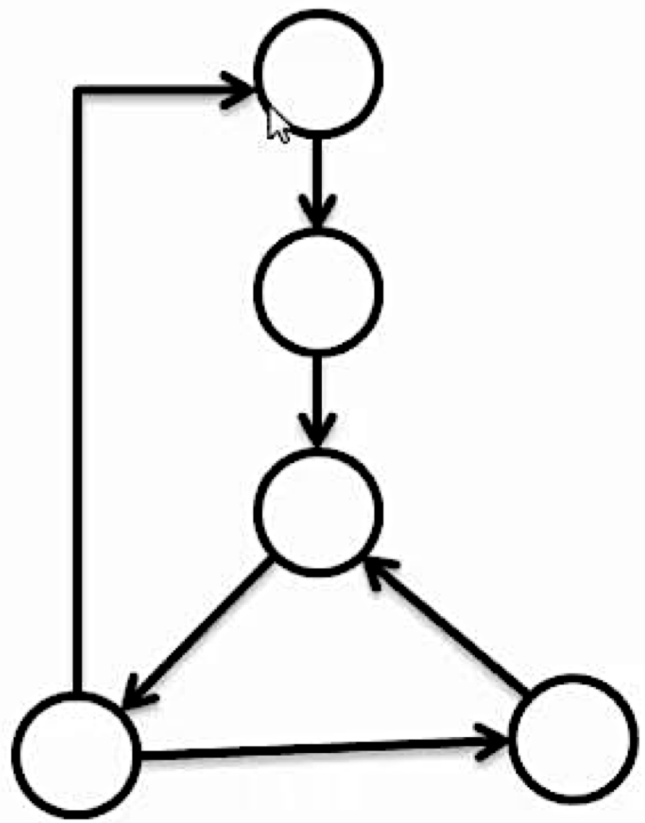
- Strongly connected graph

  A *directed graph* is **strongly connected** if there is a path from any vertex a to any other vertex b of the graph.

- Weakly Connected graph

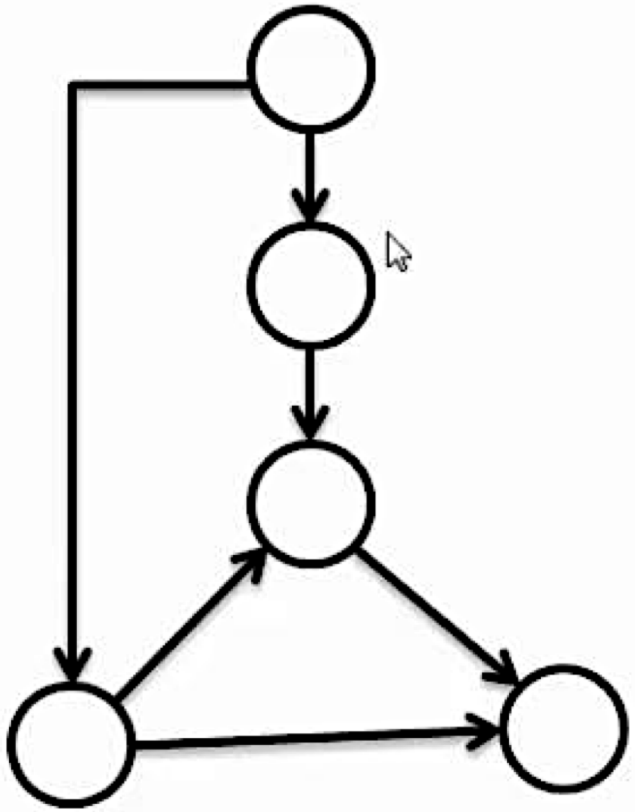
Strongly Connected Graph

In a strongly connected graph the nodes can be visited by a single path.



Weakly Connected Graph

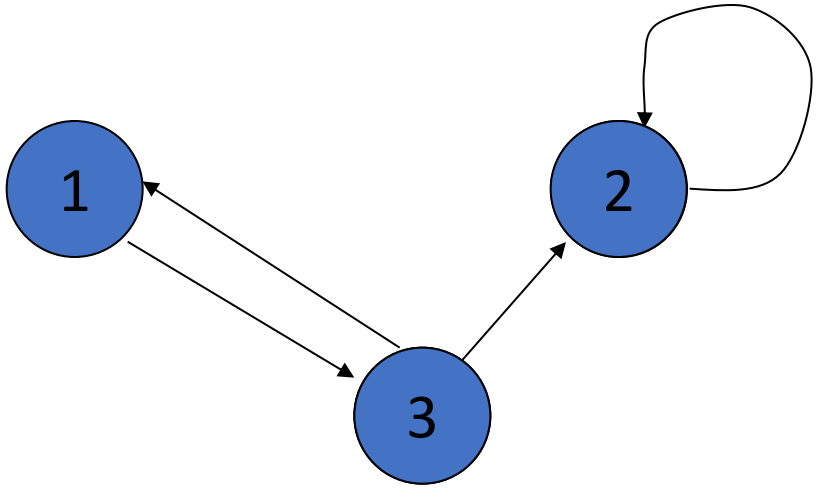
In a weakly connected graph the nodes cannot be visited by a single path.



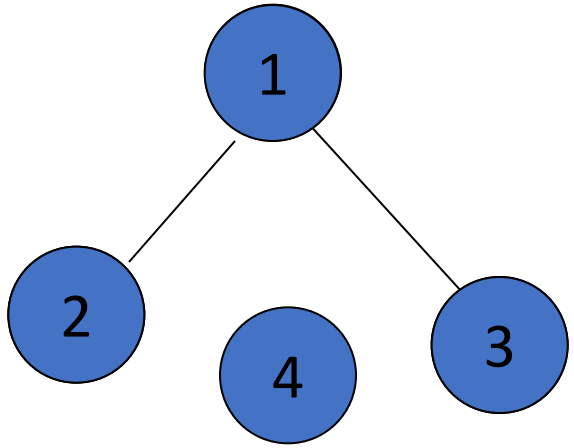
Representing graphs

- Adjacency matrix:
  - When graph is dense
  - $|E|$ close to $|V|^2$
- Adjacency lists:
  - When graph is sparse
  - $|E| << |V|^2$

Examples



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



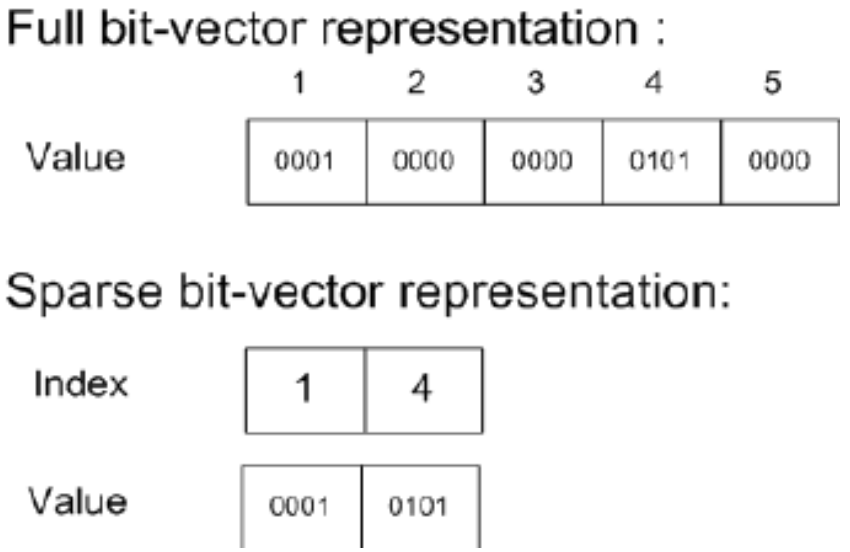
	1	2	3	4
1	0	1	1	0
2	1	0	0	0
3	1	0	0	0
4	0	0	0	0

Adjacency Matrix

- Matrix of size |V| x |V|
  - Each row (column) j correspond to a distinct vertex j
- “1” in cell < i, j > if there is exists an edge <i,j>
- Otherwise, “0”

Adjacency matrix features

- Storage complexity: $O(|V|^2)$
  - But can use bit-vector representation
- In-degree of X: Sum along column X $O(|V|)$
- Out-degree of X: Sum along row X $O(|V|)$
- Very simple, good for small graphs
  - Edge existence query: O(1)



But, ….

- Many graphs in practical problems are sparse
  - Not many edges --- not all pairs x,y have edge x→y
- Matrix representation demands too much memory
- We want to reduce memory footprint
  - Use sparse matrix techniques

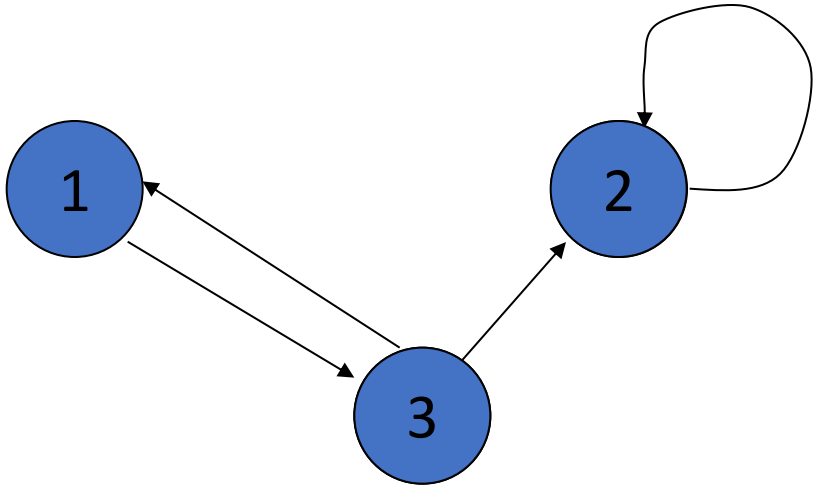
Adjacency List

- An array Adj[ ] of size |V|
- Each cell holds a list for associated vertex
- Adj[u] is list of all vertices adjacent to u
  - List does not have to be sorted

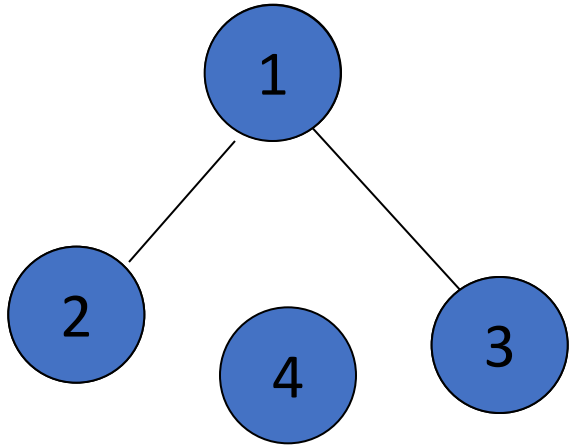
Undirected graphs:

- Each edge is represented twice

Examples



1	→ 3
2	→ 2
3	→ 1 → 2



1	→ 2 → 3
2	→ 1
3	→ 1
4	→

Adjacency list features

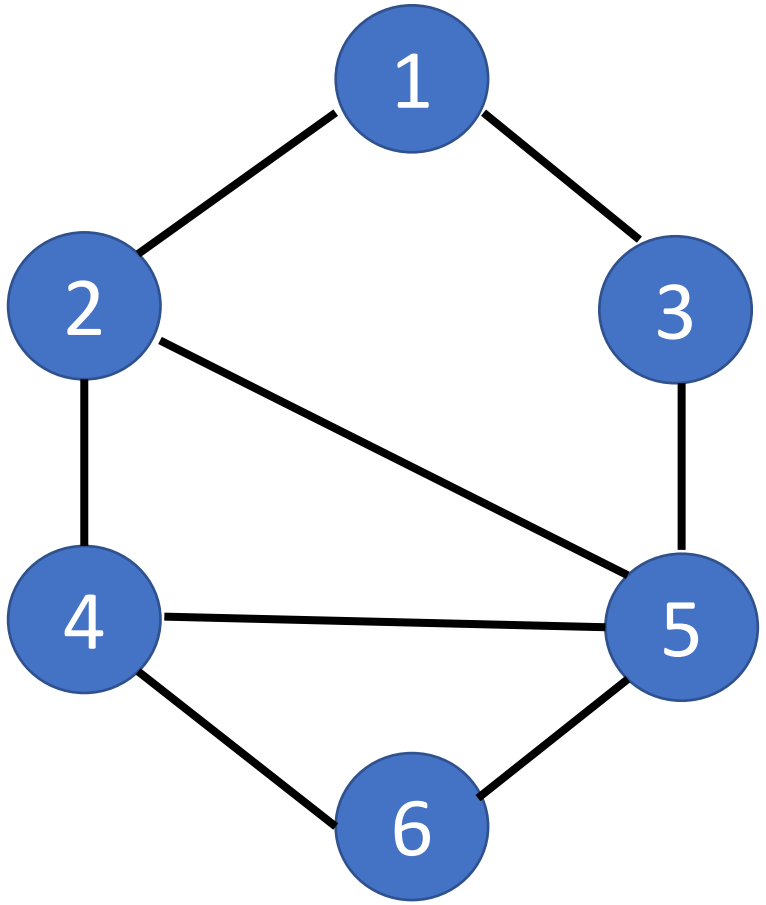
- Storage Complexity:
  - O(|V| + |E|)
  - In undirected graph: O(|V|+2*|E|) = O(|V|+|E|)
- Edge query check:
  - O(|V|) in worst case
- Degree of node X:
  - Out degree: Length of Adj[X] O(|V|) calculation
  - In degree: Check all Adj[] lists O(|V|+|E|)
  - Can be done in O(1) with some auxiliary information!

Graph Traversals

- Breadth-first search (BFS)
- Depth-first search (DFS)

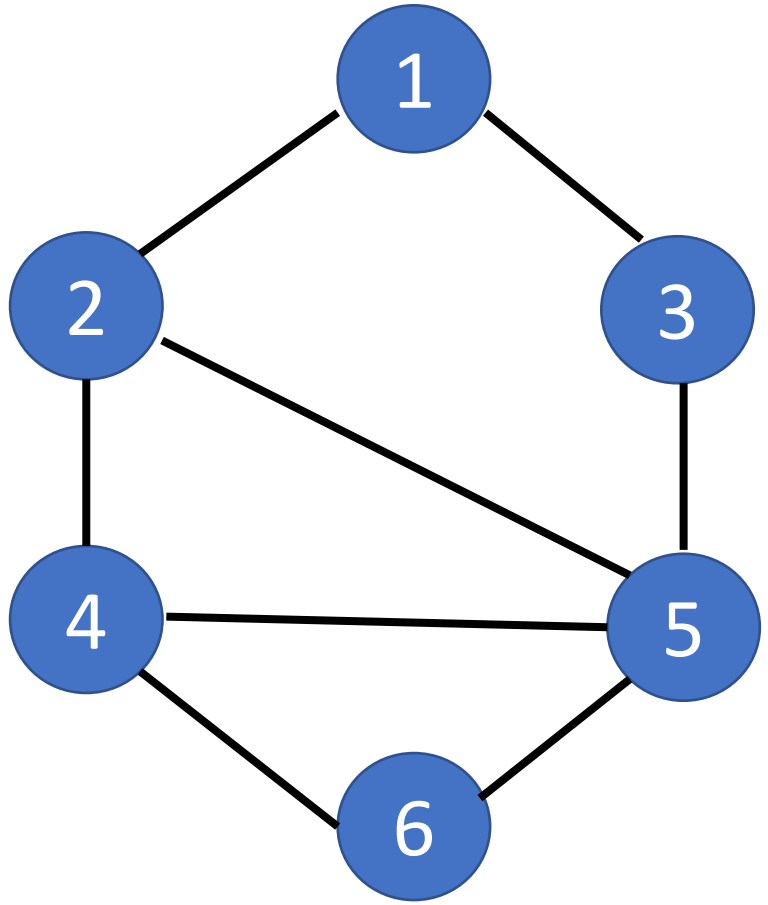
```
void Graph::BFS(int s)
{
    // Mark all the vertices as not visited
    bool *visited = new bool[V];
    for(int i = 0; i < V; i++)
        visited[i] = false;

    // Create a queue for BFS
    list<int> queue;

    // Mark the current node as visited and enqueue it
    visited[s] = true;
    queue.push_back(s);

    // 'i' will be used to get all adjacent
    // vertices of a vertex
    list<int>::iterator i;

    while(!queue.empty())
    {
        // Dequeue a vertex from queue and print it
        s = queue.front();
        cout << s << " ";
        queue.pop_front();

        // Get all adjacent vertices of the dequeued
        // vertex s. If a adjacent has not been visited,
        // then mark it visited and enqueue it
        for (i = adj[s].begin(); i != adj[s].end(); ++i)
        {
            if (!visited[*i])
            {
                visited[*i] = true;
                queue.push_back(*i);
            }
        }
    }
}
```

BFS



BFS

We did this before for binary trees
Unless in graph there might be cycles!
We may end up in an infinite loop.

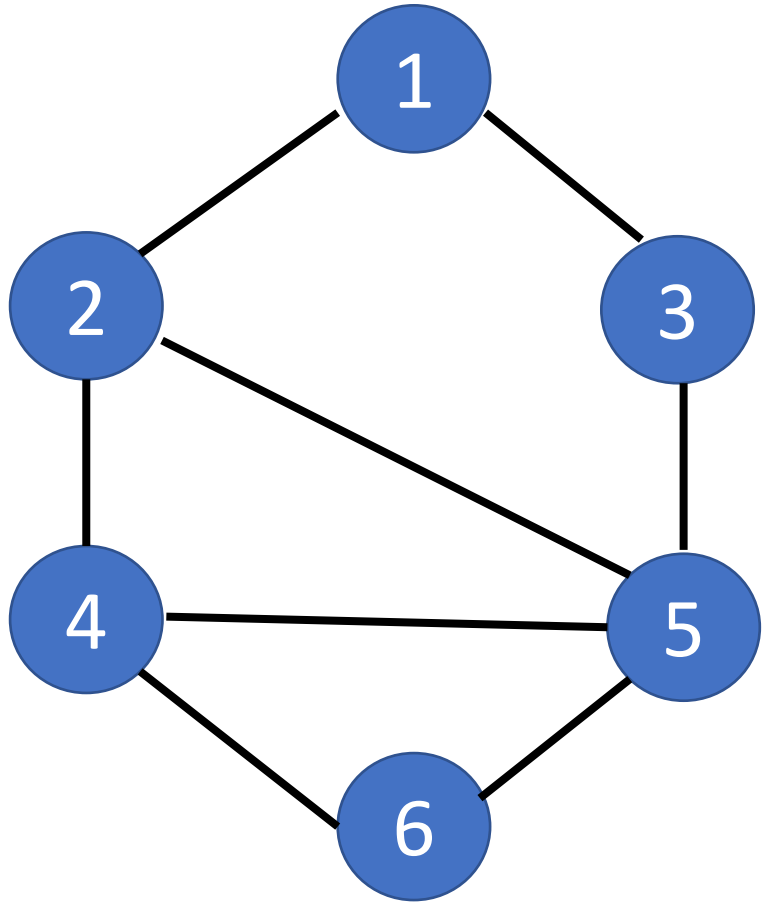


BFS

Visited

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

Queue:



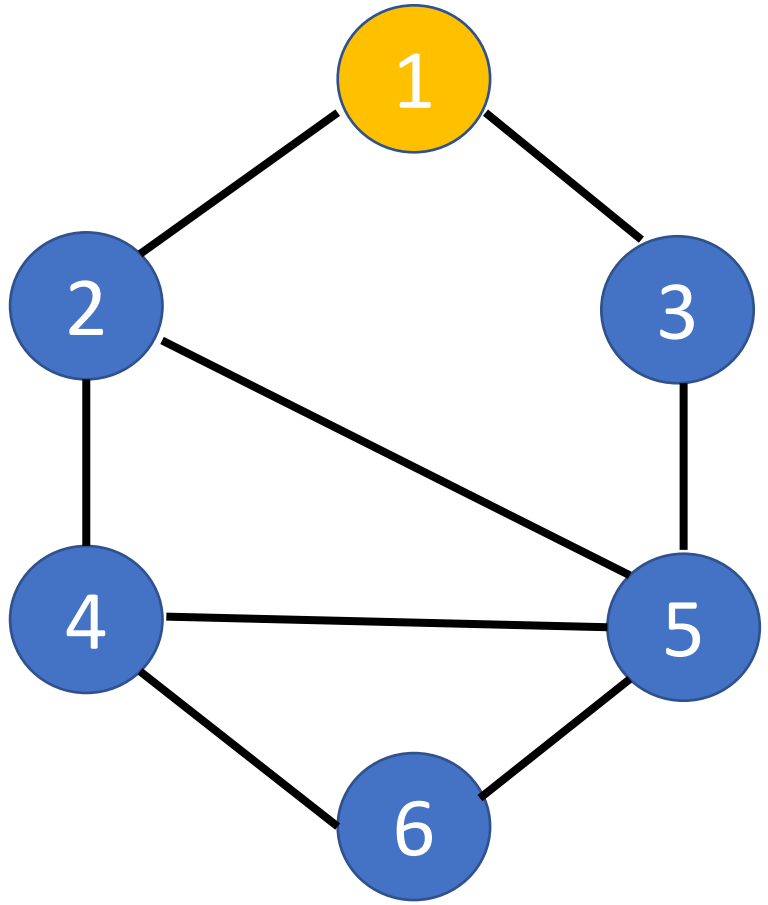
BFS

Visited

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

Queue: **1**

Print: **1**



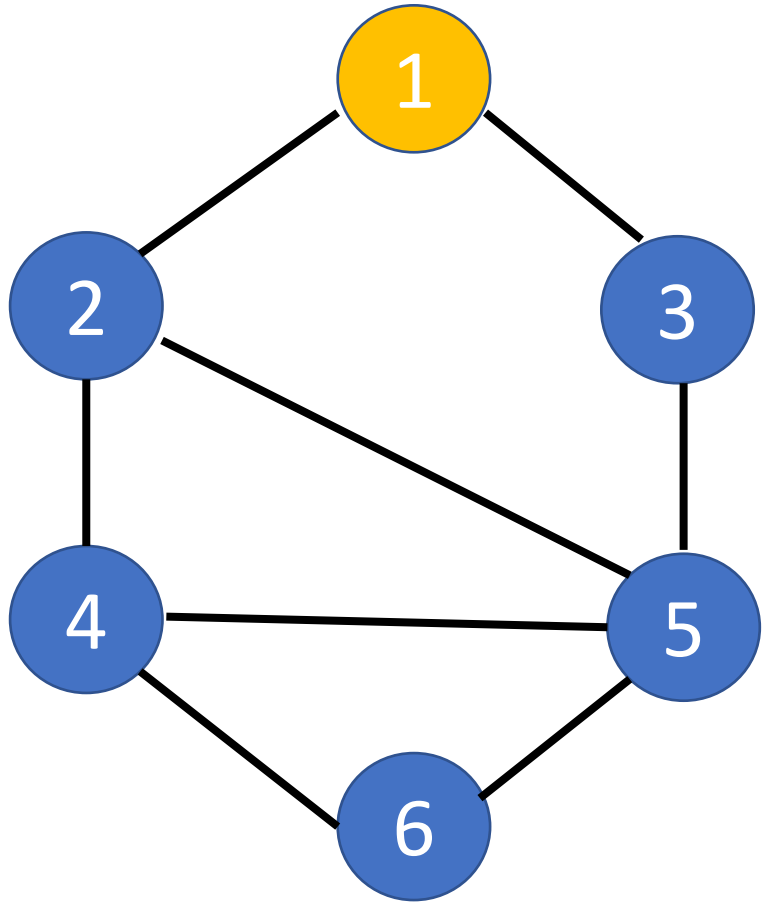
BFS

Visited

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

Queue:

Print: 1



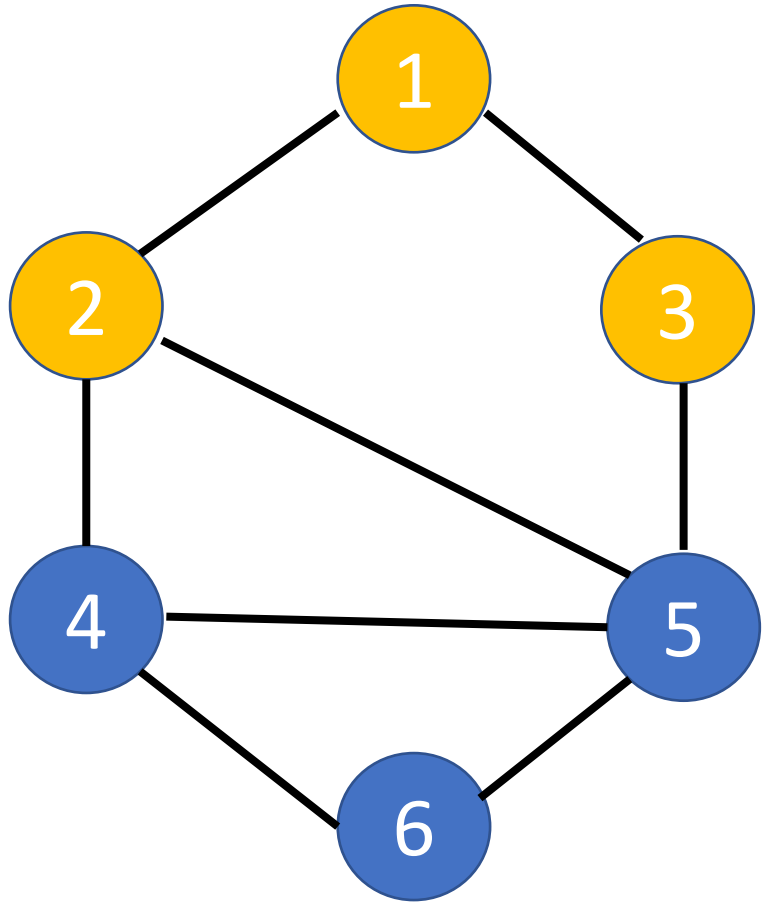
BFS

Visited

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

Queue: 2 3

Print: 1



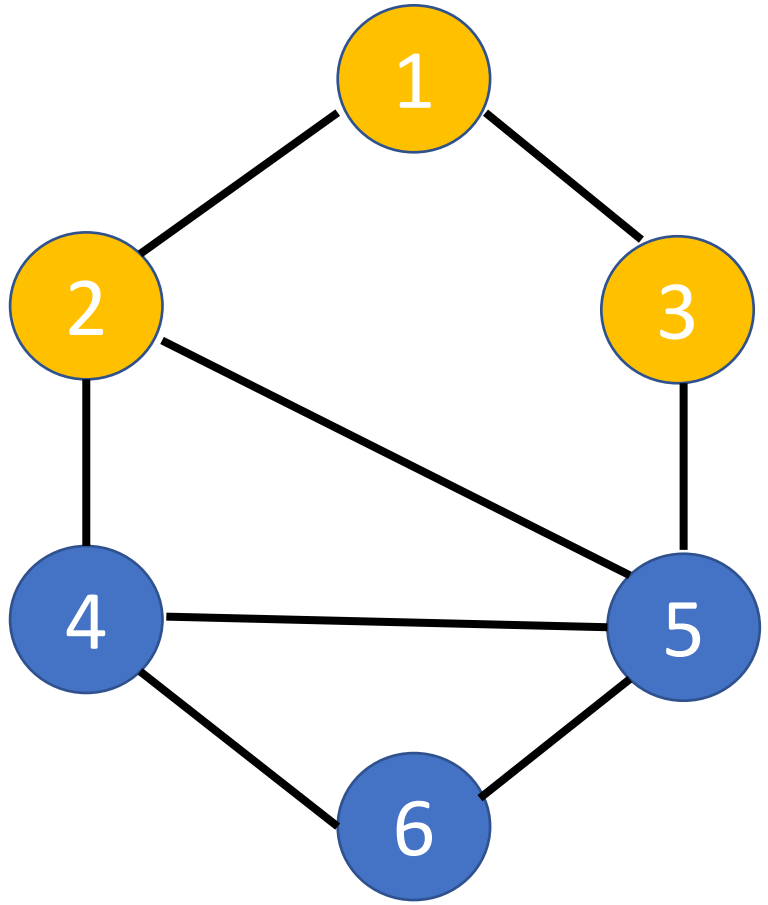
BFS

Visited

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

Queue: 3

Print: 1 2



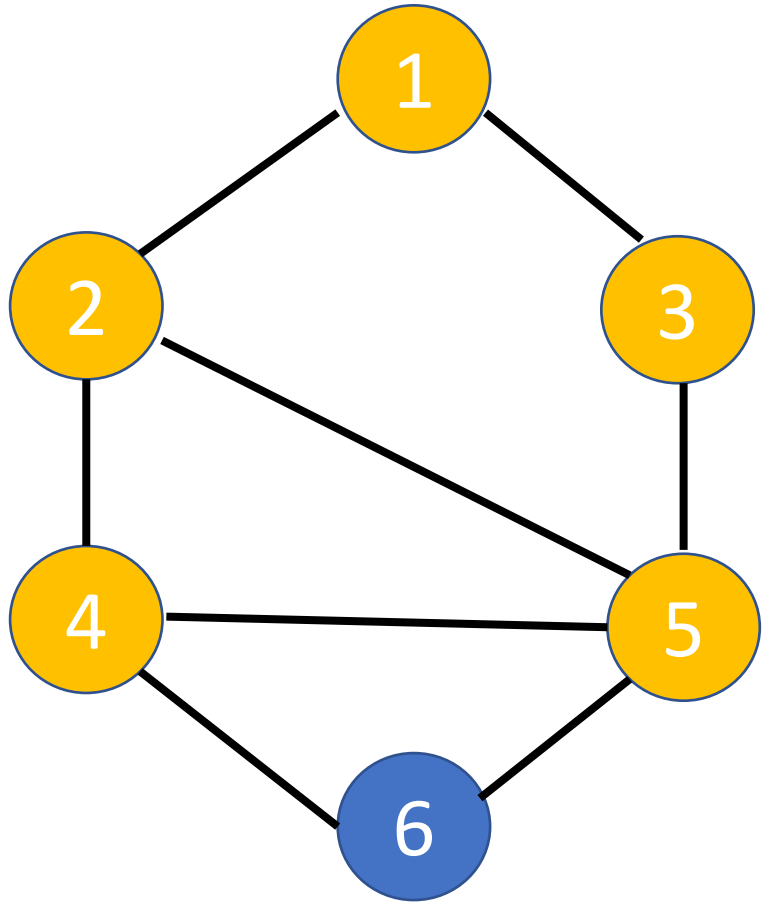
BFS

Visited

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

Queue: 3 4 5

Print: 1 2



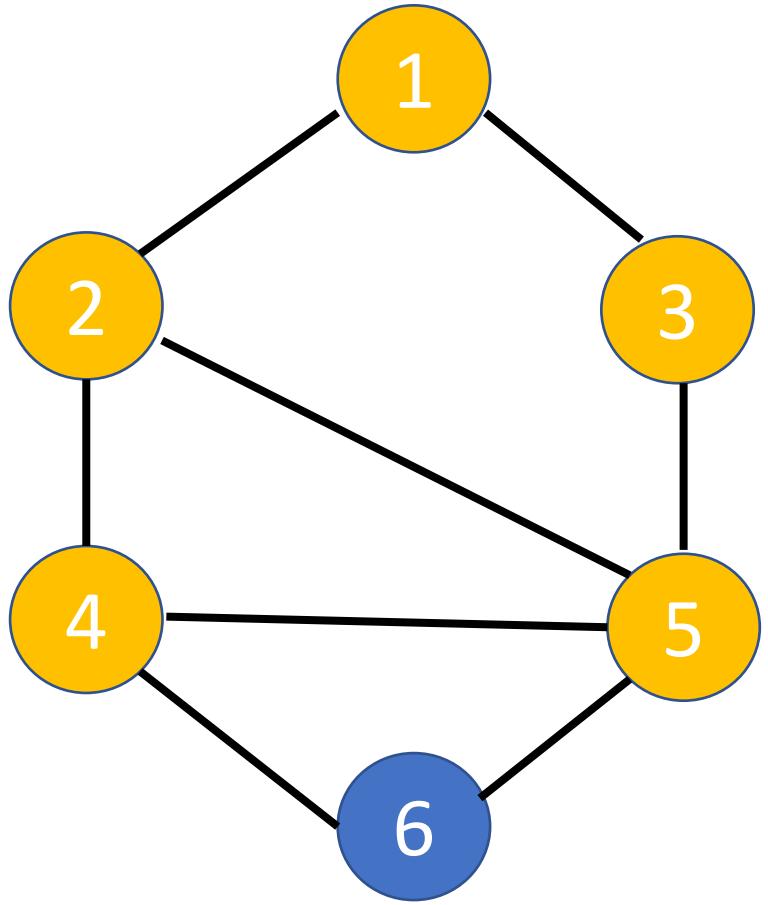
BFS

Visited

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

Queue: 4 5

Print: 1 2 3



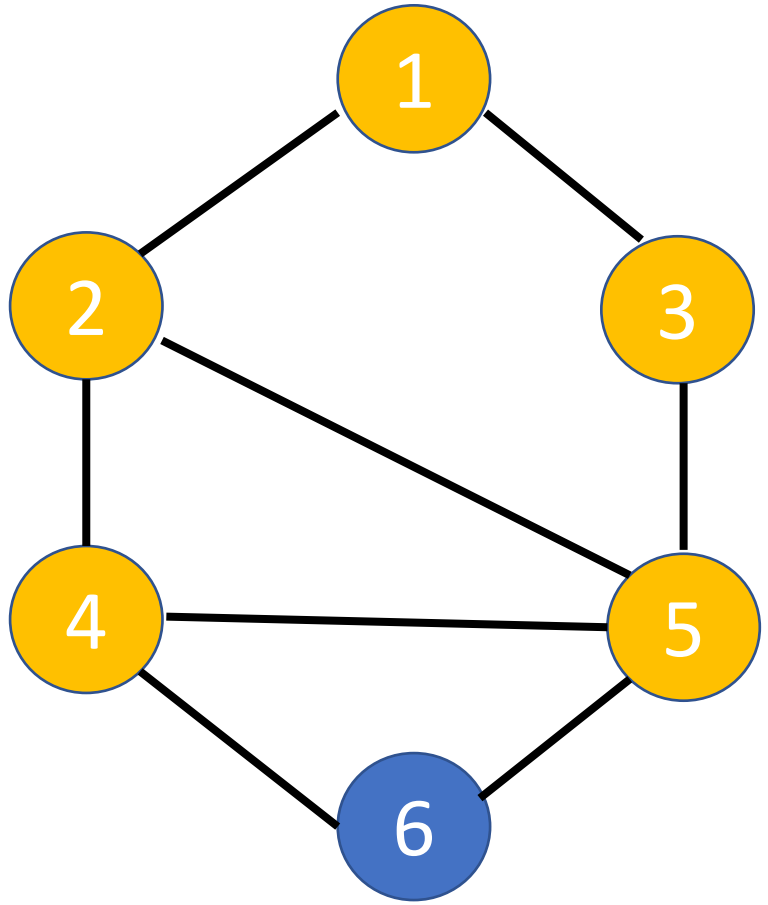
BFS

Visited

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

Queue: 5

Print: 1 2 3 4



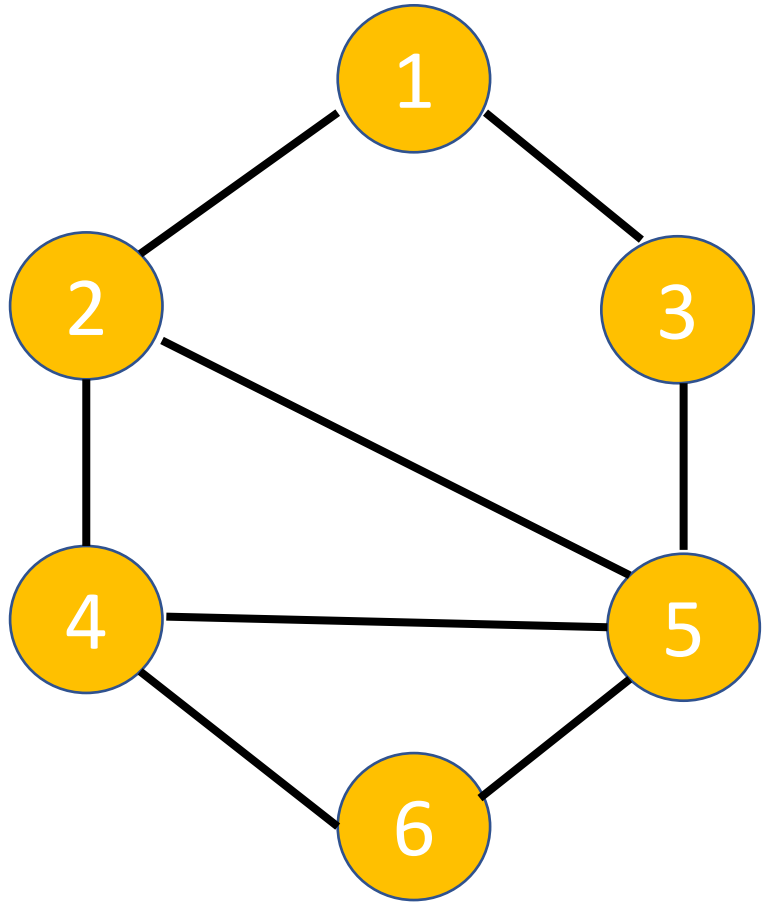
BFS

Visited

1	2	3	4	5	6
1	1	1	1	1	1

Queue: 5 6

Print: 1 2 3 4



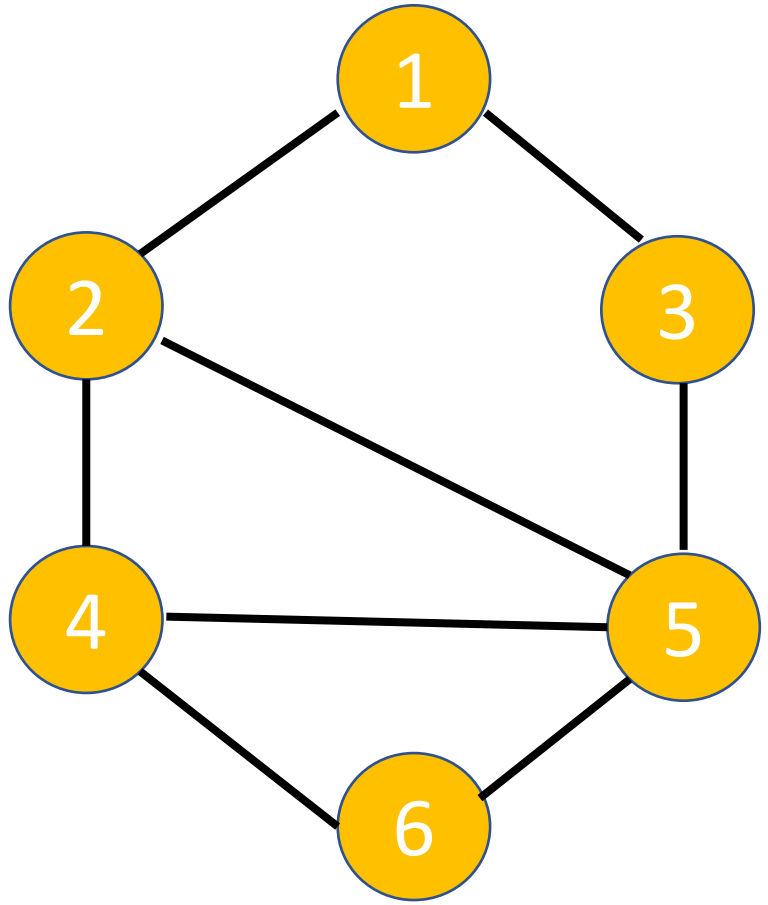
BFS

Visited

1	2	3	4	5	6
1	1	1	1	1	1

Queue: 6

Print: 1 2 3 4 5



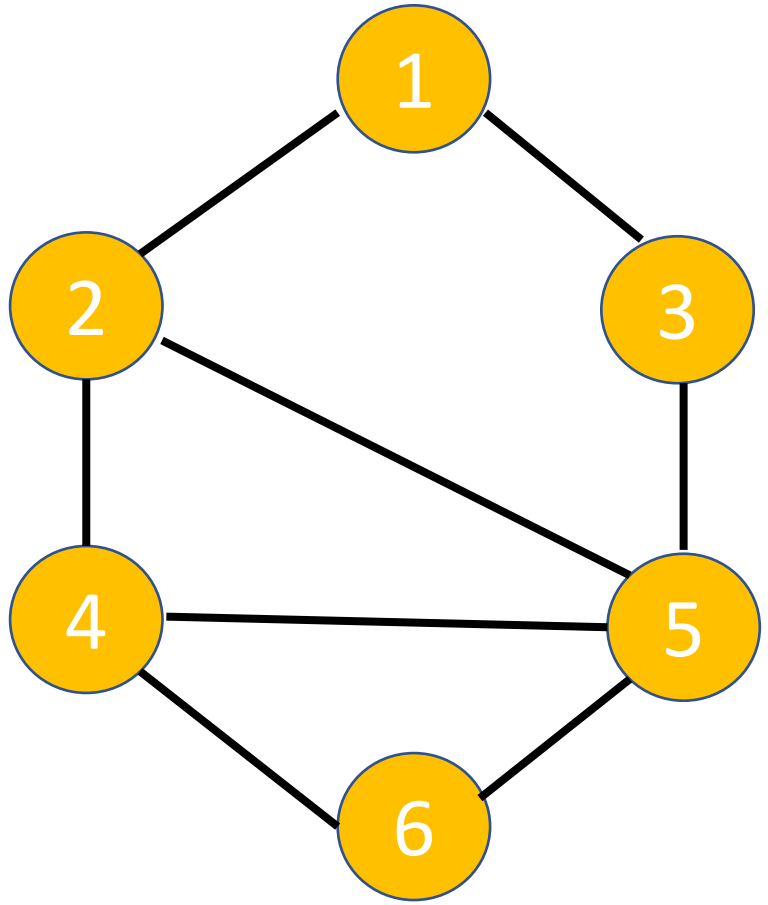
BFS

Visited

1	2	3	4	5	6
1	1	1	1	1	1

Queue:

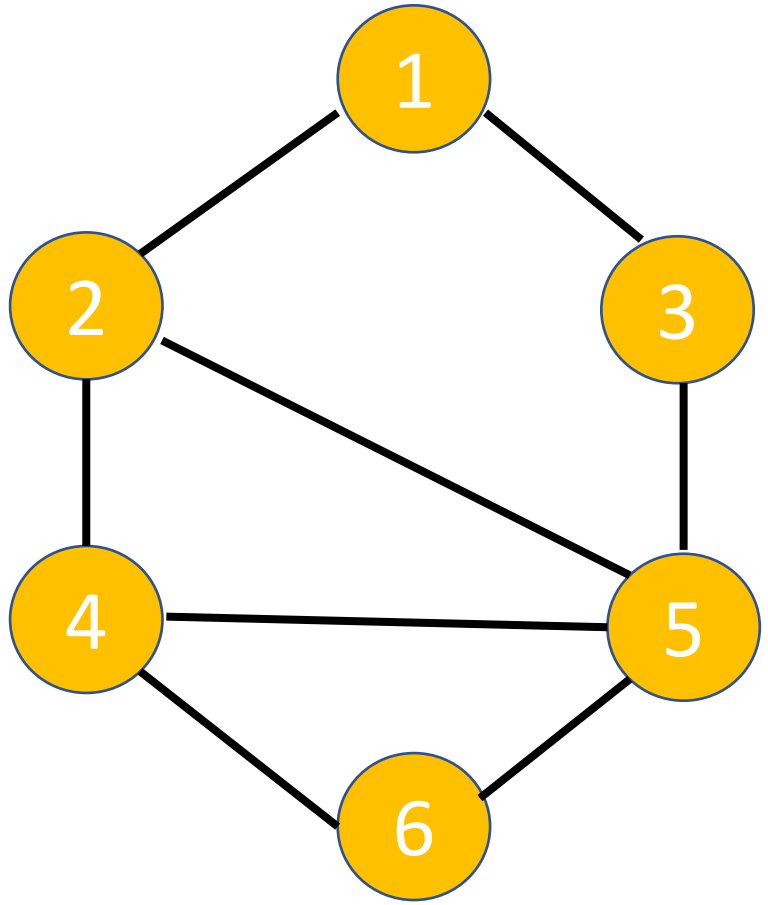
Print: 1 2 3 4 5 6



BFS - Complexity

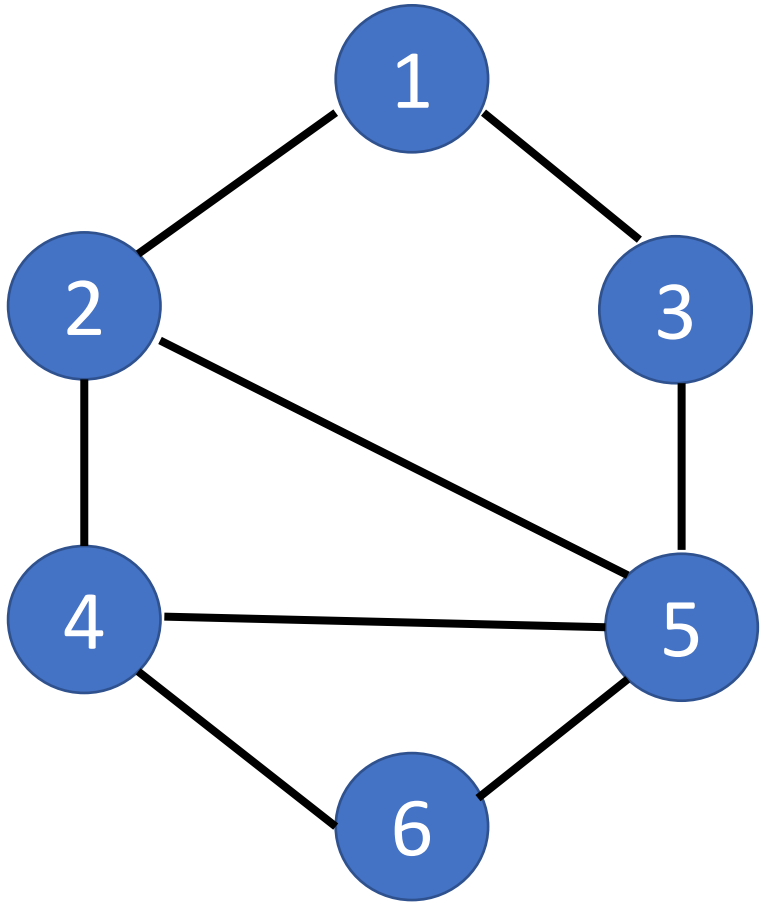
O (v + E)

V = Vertices
E = Edges

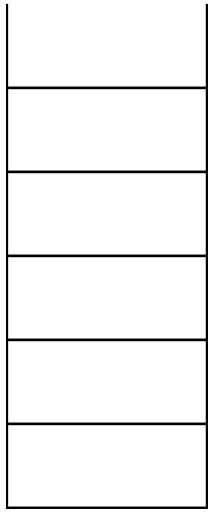


```
void Graph::DFSUtil(int v, bool visited[])
{
    // Mark the current node as visited and
    // print it
    visited[v] = true;
    cout << v << " ";

    // Recur for all the vertices adjacent
    // to this vertex
    list<int>::iterator i;
    for (i = adj[v].begin(); i != adj[v].end(); ++i)
        if (!visited[*i])
            DFSUtil(*i, visited);
}

// DFS traversal of the vertices reachable from v.
// It uses recursive DFSUtil()
void Graph::DFS(int v)
{
    // Mark all the vertices as not visited
    bool *visited = new bool[V];
    for (int i = 0; i < V; i++)
        visited[i] = false;

    // Call the recursive helper function
    // to print DFS traversal
    DFSUtil(v, visited);
}
```

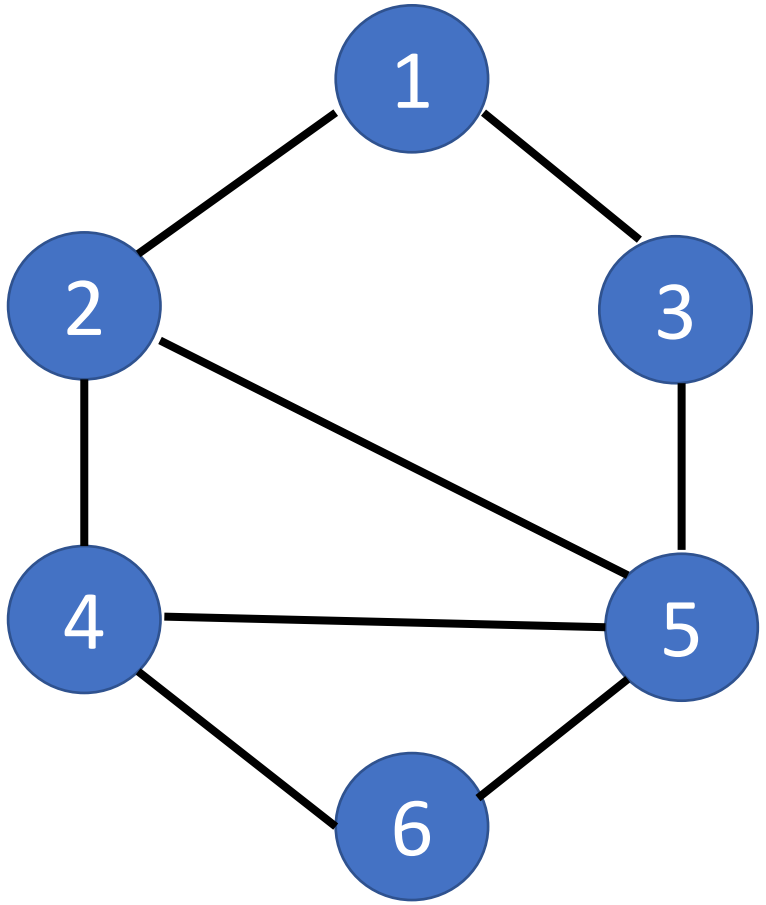
DFS



DFS

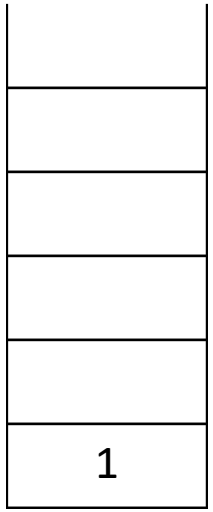


Stack

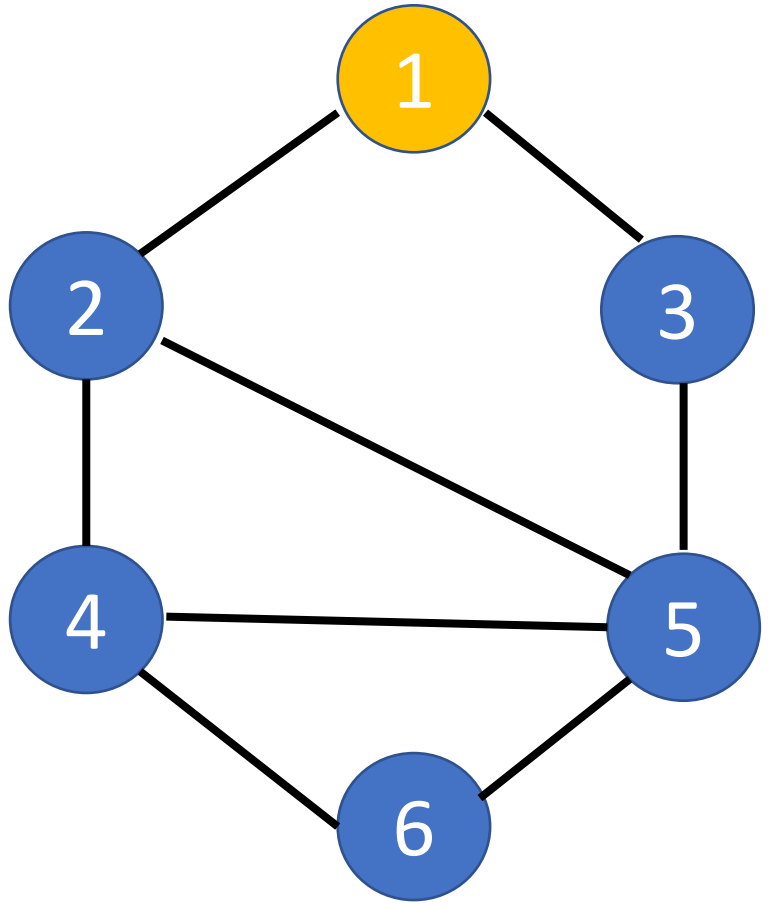


Print:

DFS

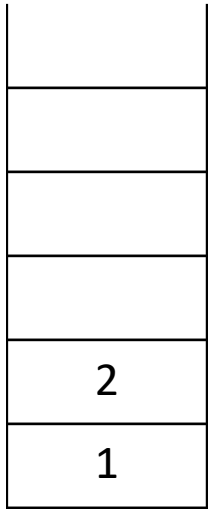


Stack

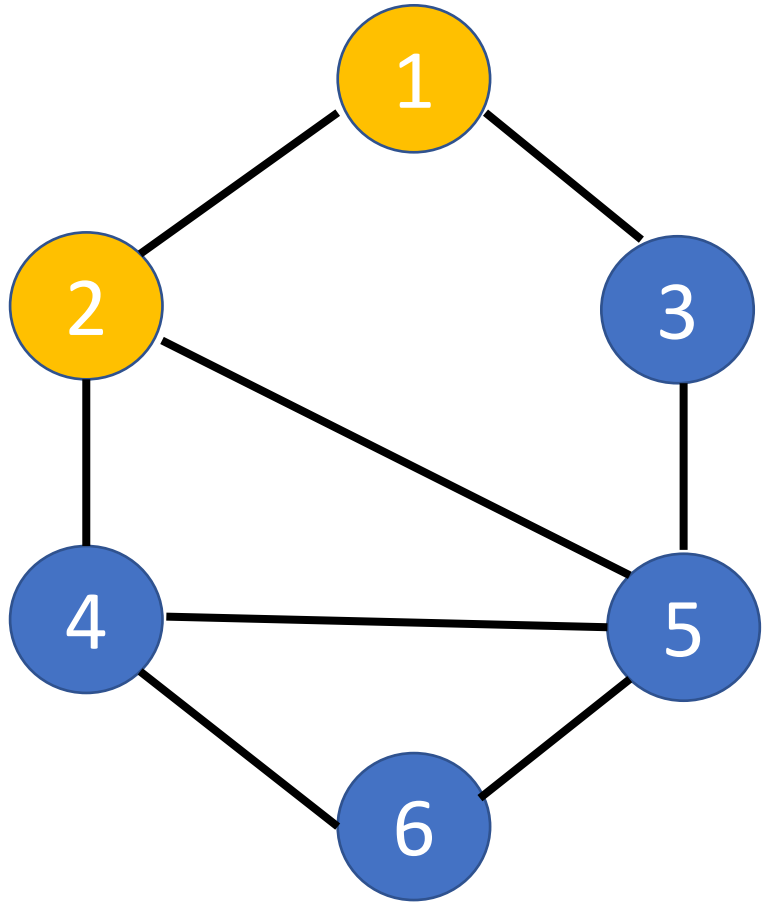


Print: **1**

DFS

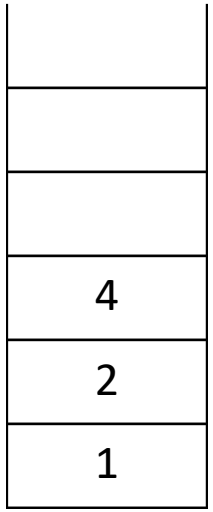


Stack

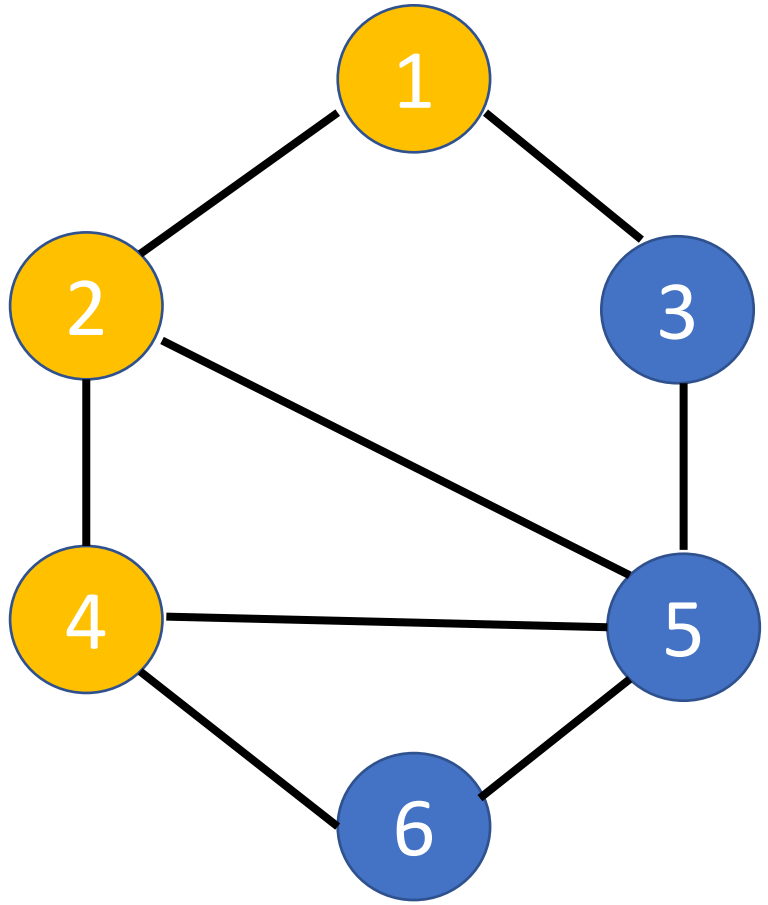


Print: **1 2**

DFS



Stack

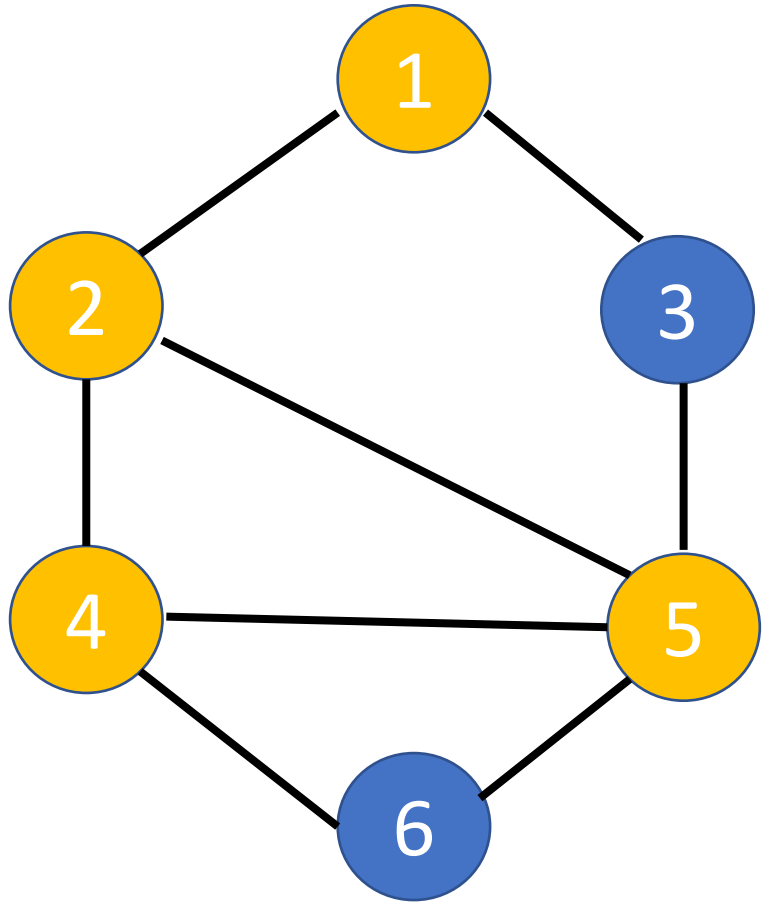


Print: 1 2 4

DFS

5
4
2
1

Stack

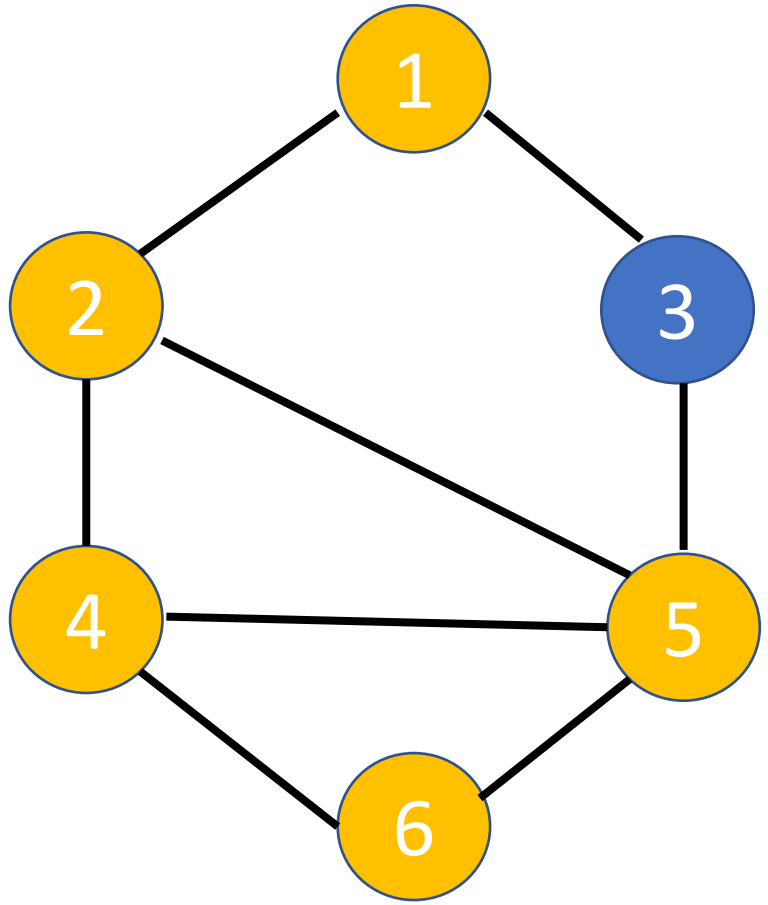


Print: 1 2 4 5

DFS

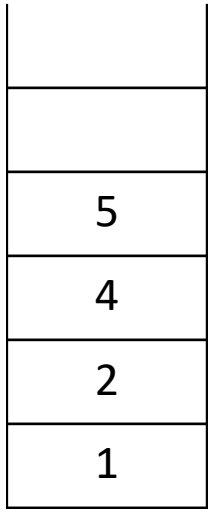
6
5
4
2
1

Stack

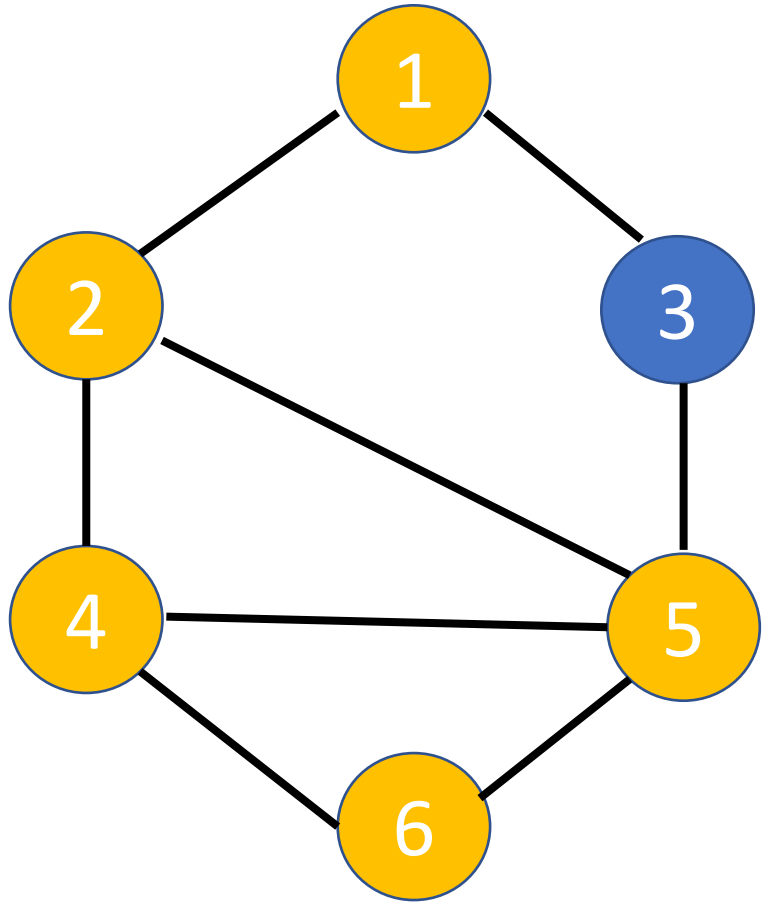


Print: 1 2 4 5 6

DFS



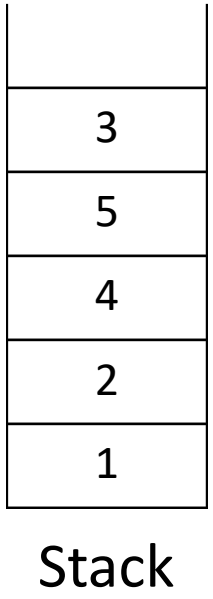
Stack

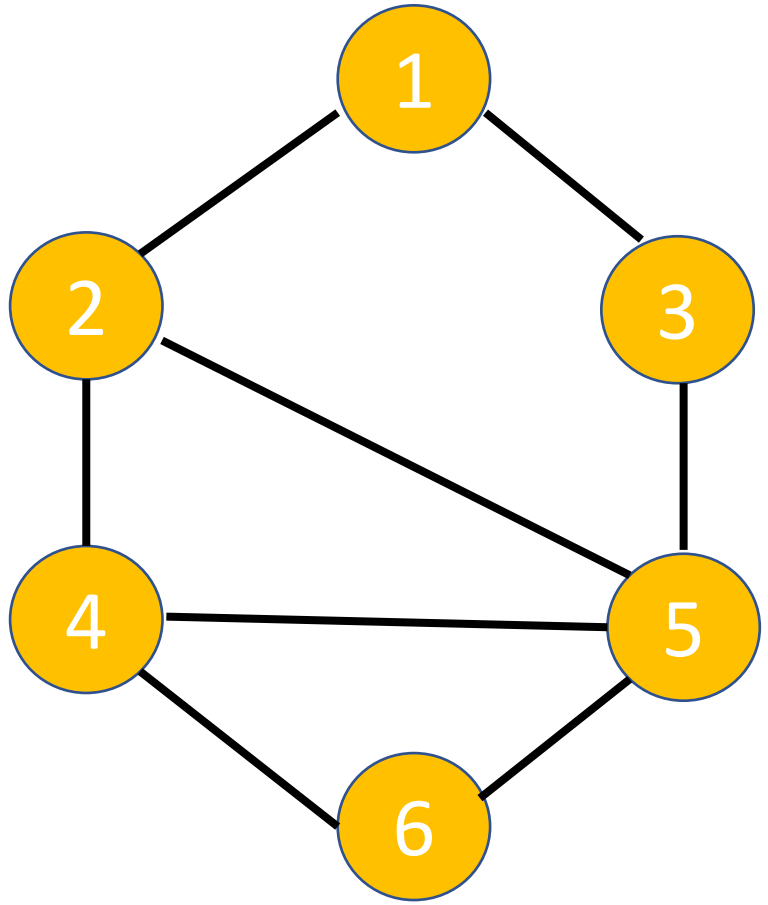


All the nodes connected with 6 have already visited!

Print: 1 2 4 5 6

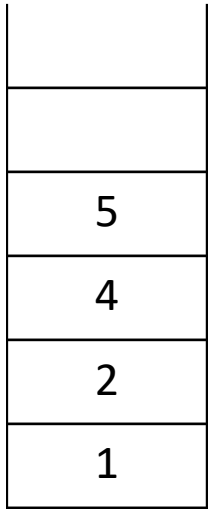
DFS



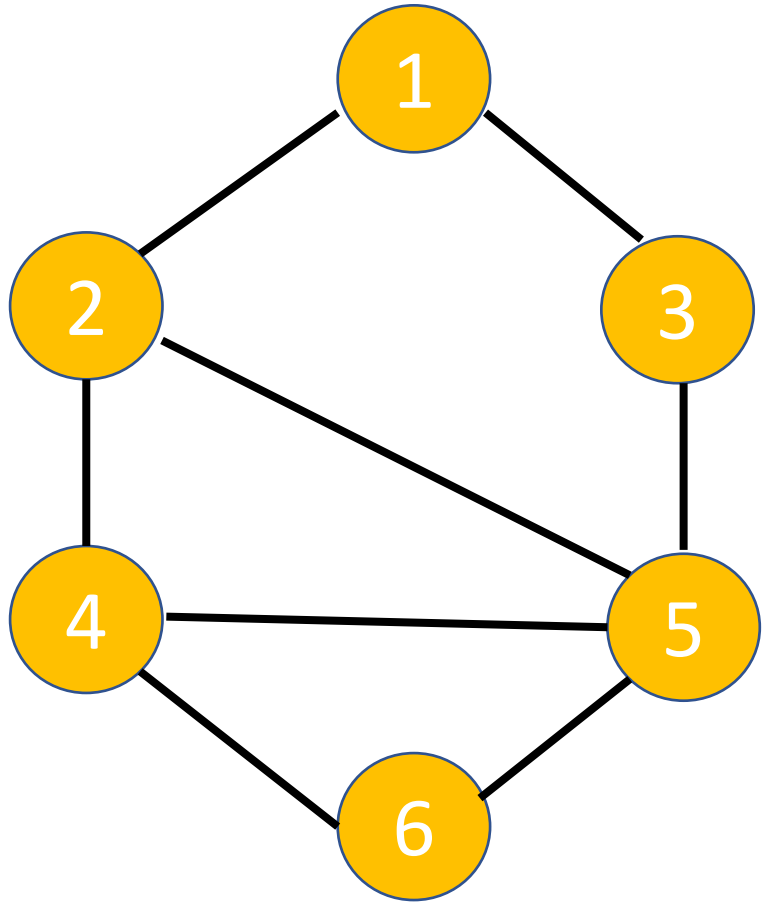


Print: **1 2 4 5 6 3**

DFS

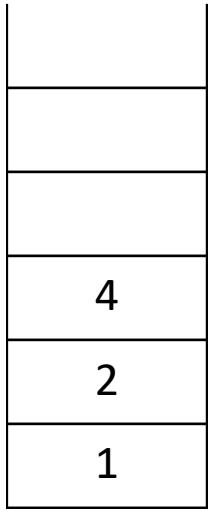


Stack

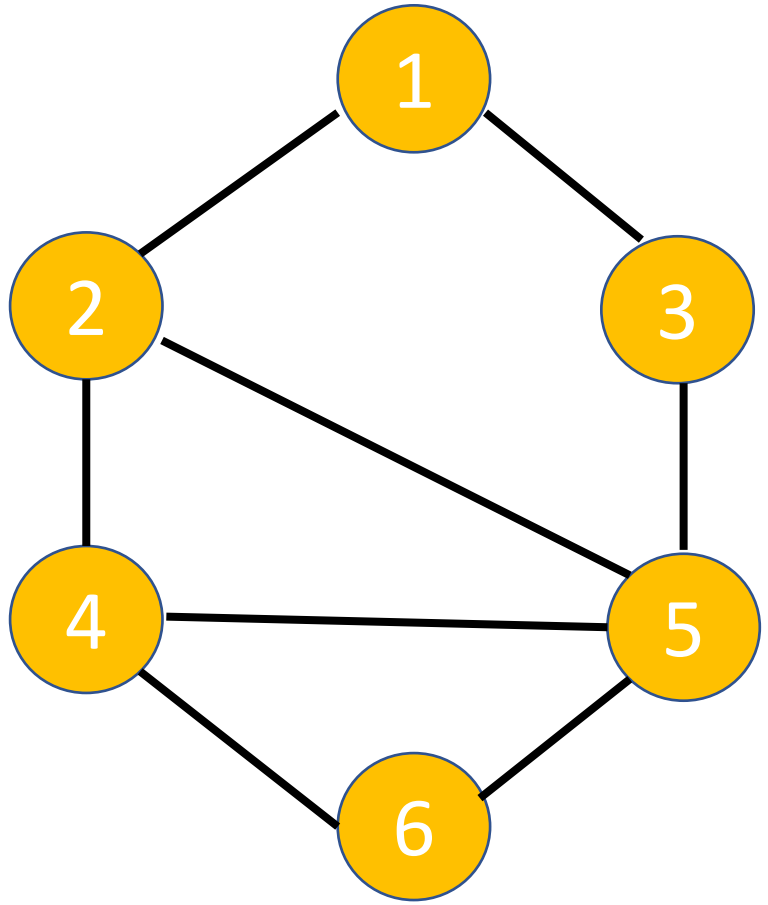


Print: 1 2 4 5 6 3

DFS

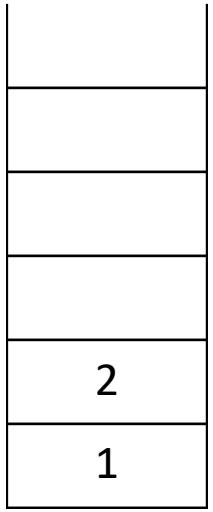


Stack

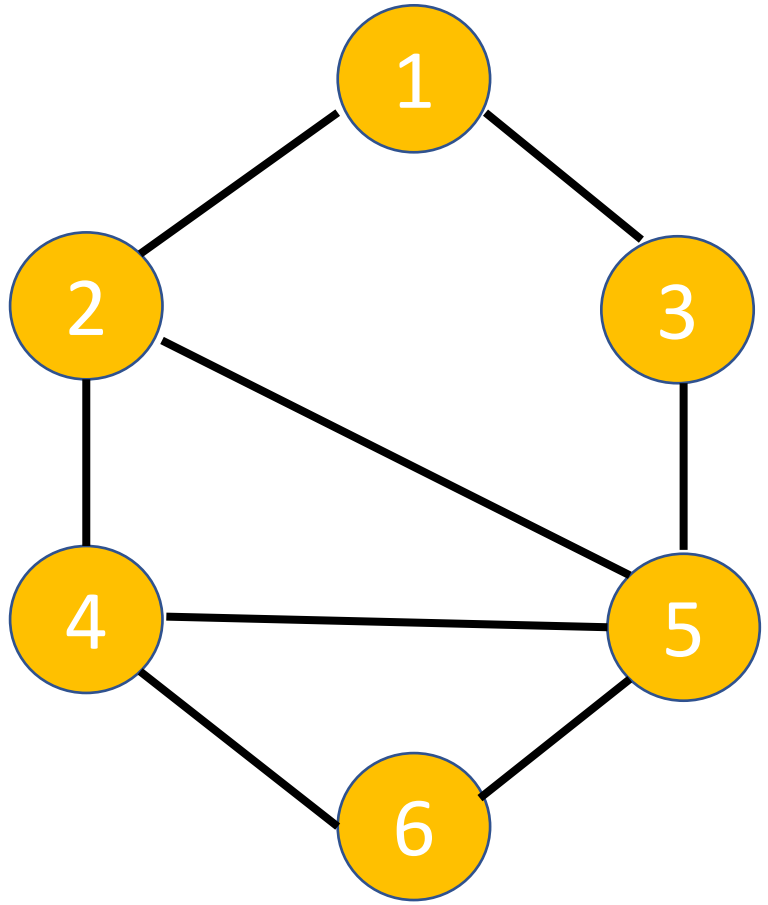


Print: **1 2 4 5 6 3**

DFS

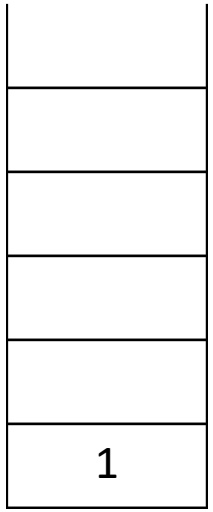


Stack

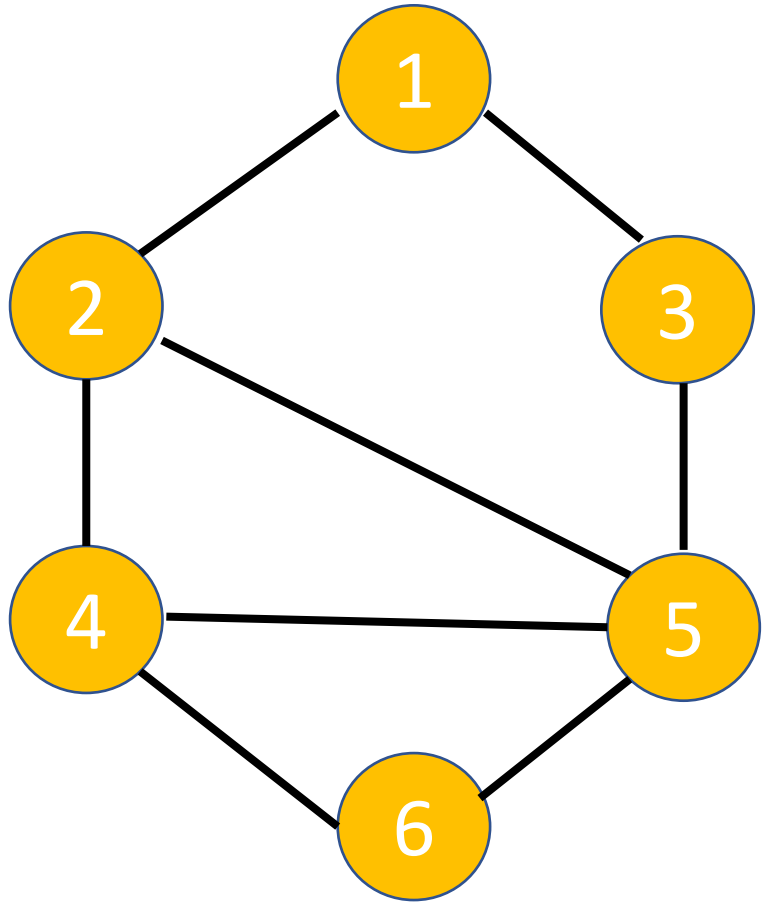


Print: **1 2 4 5 6 3**

DFS

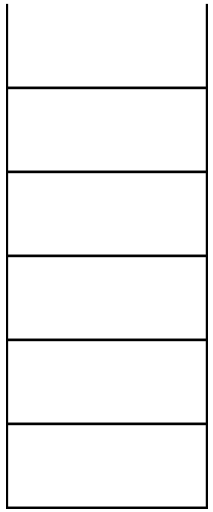


Stack

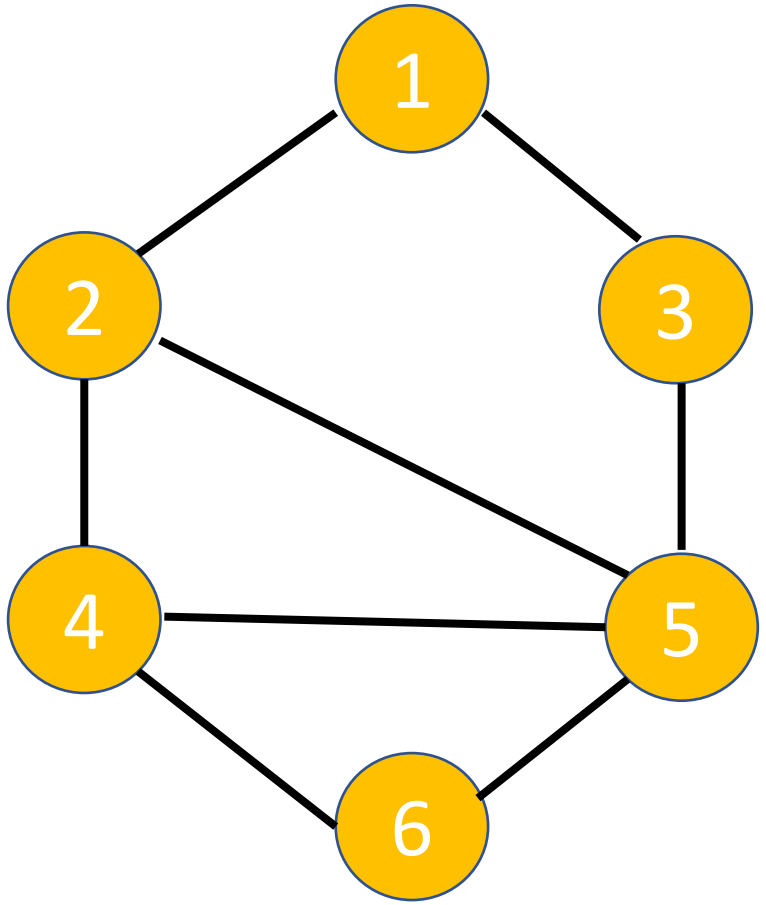


Print: **1 2 4 5 6 3**

DFS



Stack

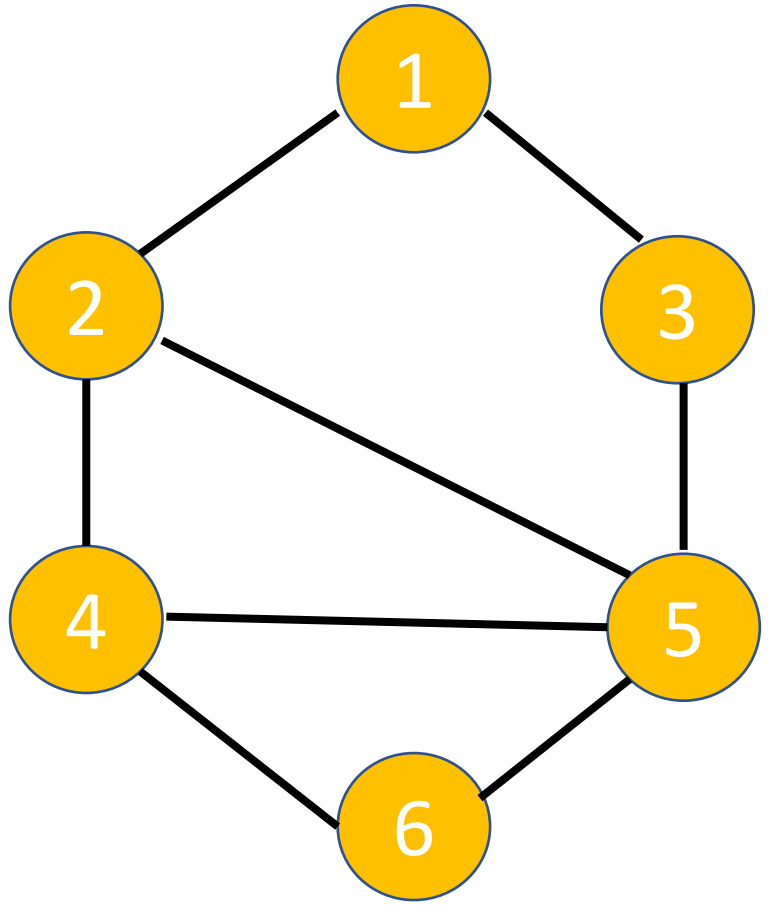


Print: **1 2 4 5 6 3**

DFS - Complexity

O (v + E)

V = Vertices
E = Edges



Applications of Graph

- Social Network
- Endless
- 6 Degree of Separation!!

- https://www.youtube.com/watch?v=TcxZSmzPw8k