1) BFS of a graph

```
BFS(graph G, vertex s):
    mark s = True
    enqueue s to Queue
    while Queue != []:
        v = dequeue()
        for w in G.adj[v]:
            if not visited[w]:
                visited[w] = True
                enqueue(w)
    return visited
```

2) DFS of a graph

```
DFS(graph G, vertex s):
    visited = [False] * len(G.vertices)
    stack = []
    stack.append(s)
    while stack != []:
        v = stack.pop()
        if not visited[v]:
            visited[v] = True
            for w in G.adj[v]:
                if not visited[w]:
                    stack.append(w)
    return visited
```

3) Detect cycle in a directed graph

```
DFS(graph G, vertex s):
    visited = [False] * len(G.vertices)
    stack = []
    stack.append(s)
    while stack != []:
        v = stack.pop()
        if not visited[v]:
            visited[v] = True
            for w in G.adj[v]:
                if not visited[w]:
                    stack.append(w)
    return visited
```

4) Topological Sort

DFS Approach

Step: Choose some arbitrary vertex of graph and keep visiting every vertex that you can reach from current vertex. Once you can't go anywhere, mark current vertex as visited and add it to front of topsort. Then, choose another vertex that hasn't been visited yet and repeat process.

```
TopSort(graph G):
    for v in G.vertices:
        visited[v] = False
    topsort = []
    for v in G.vertices:
        if not visited[v]:
            DFS_visit(G, v, visited, topsort)
    return topsort

DFS_visit(graph G, vertex v, visited, topsort):
    visited[v] = True
    for w in G.adj[v]:
        if not visited[w]:
            DFS_visit(G, w, visited, topsort)
    topsort.append(v)
    return
```

BFS Approach

Step: Find vertices that have no incoming edges. Add them to topsort. Remove these vertices and their outgoing edges from graph. Repeat process until there are no vertices left.

```
TopSort(graph G):
    while G.vertices != []:
        for v in G.vertices:
            if len(G.adj[v]) == 0:
                topsort.append(v)
                G.remove_vertex(v)
    return topsort
```

5) Detect cycle in undirected graph

DFS Approach

Step: Choose some arbitrary vertex of graph and keep visiting every vertex that you can reach from current vertex. Once you can't go anywhere, mark current vertex as visited and add it to front of topsort. Then, choose another vertex that hasn't been visited yet and repeat process.

Note: If you encounter a vertex that you've already visited, then there is a cycle. Keep track of parent vertex of current vertex. If you encounter a vertex that you've already visited, check if it's the parent of current vertex. If it is, then there is no cycle. If it isn't, then there is a cycle.

```
DFS(graph G):
    for v in G.vertices:
        visited[v] = False
    for v in G.vertices:
        if not visited[v]:
            DFS_visit(G, v, None, visited)
    return True

DFS_visit(graph G, vertex v, vertex parent, visited):
    visited[v] = True
    for w in G.adj[v]:
        if w == parent:
            continue
        if visited[w]:
            return False
        else:
            if not DFS_visit(G, w, v, visited):
                return False
    return True
```

6) Cycle, and bipartite

- cycle is the condition the graph should satisfy to be a directed graph with number of vertices equal to number of edges in the tree.
- bipartite condition is checking if there is a cycle.
- graph with no cycle, if graph has no edge, so each vertex belongs to the bipartite set. Or, graph has one edge.
- If graph has cycle, then it is bipartite, then edge count must be 2 or bigger or equal to cycle length.
- Then, all edges are equal, which is not true that adjacently find cycle of odd length.
- graph is bipartite if and only if it has no cycle of odd length.
- In graph $G = (V, E)$ if $v \in V$, then v is a cut vertex, if bipartite graph $G/\{v\}$ is a disjoint union of two connected components.
- In graph $G = (V, E)$ if $v \in V$, then v is a cut vertex if bipartite graph $G/\{v\}$ is a disjoint union of two connected components.

7) Find-Set Algorithm

```
Find_set(x):
    while x.parent != x:
        x = x.parent
    return x
```

8) Union Algorithm (Topological Sorting (DFS))

$O(V+E)$ time.
$O(V)$ space.

9) Union Algorithm (Topological Sorting (BFS))

$O(V+E)$ time.
$O(V)$ space.

10) Dijkstra's Algorithm

```
Dijkstra(graph G, vertex s):
    for v in G.vertices:
        dist[v] = float('inf')
    dist[s] = 0
    while Queue != []:
        u = Queue.pop()
        for v in G.adj[u]:
            if dist[v] > dist[u] + G.w(u, v):
                dist[v] = dist[u] + G.w(u, v)
                Queue.push(v)
    return dist
```

11) Dijkstra's Algorithm

- Dijkstra's algorithm is a greedy algorithm that finds the shortest path from a source vertex to all other vertices in a directed graph.
- Dijkstra's algorithm works by maintaining a priority queue of vertices, where the priority of each vertex is the shortest distance from the source vertex to that vertex.
- The algorithm starts by initializing the priority queue with the source vertex and its distance from itself (which is 0). Then, it repeatedly extracts the vertex with the shortest distance from the priority queue, and updates the distances of its neighbors if a shorter path is found.
- The algorithm stops when the priority queue is empty, or when the shortest distance to the destination vertex has been found.
- Dijkstra's algorithm is a greedy algorithm because it always chooses the vertex with the shortest distance from the source vertex, and never revisits vertices that have already been visited.
- Dijkstra's algorithm can be used to find the shortest path between two vertices in a weighted graph, or to find the shortest path from a single source vertex to all other vertices in a weighted graph.
- Dijkstra's algorithm has a time complexity of $O(V + E \log V)$, where V is the number of vertices and E is the number of edges in the graph.
- Dijkstra's algorithm can be used in various applications, such as routing in computer networks, finding the shortest path between two locations in a map, and optimizing the performance of a system by minimizing the cost of resources used.

12) Bellman-Ford Algorithm

```
Bellman_Ford(graph G, vertex s):
    for v in G.vertices:
        dist[v] = float('inf')
    dist[s] = 0
    for i in range(len(G.vertices) - 1):
        for u in G.vertices:
            for v in G.adj[u]:
                if dist[v] > dist[u] + G.w(u, v):
                    dist[v] = dist[u] + G.w(u, v)
    for u in G.vertices:
        for v in G.adj[u]:
            if dist[v] > dist[u] + G.w(u, v):
                return False
    return True
```

13) Shortest Path in DAG

```
Shortest_Path(graph G, vertex s):
    for v in G.vertices:
        dist[v] = float('inf')
    dist[s] = 0
    topsort = TopSort(G)
    for u in topsort:
        for v in G.adj[u]:
            if dist[v] > dist[u] + G.w(u, v):
                dist[v] = dist[u] + G.w(u, v)
    return dist
```

14) Shortest Path in Unweighted Graph

```
Shortest_Path(graph G, vertex s):
    for v in G.vertices:
        dist[v] = float('inf')
    dist[s] = 0
    Queue = [s]
    while Queue != []:
        u = Queue.pop()
        for v in G.adj[u]:
            if dist[v] == float('inf'):
                dist[v] = dist[u] + 1
                Queue.push(v)
    return dist
```

15) Shortest Path in Weighted Graph

```
Shortest_Path(graph G, vertex s):
    for v in G.vertices:
        dist[v] = float('inf')
    dist[s] = 0
    Queue = [s]
    while Queue != []:
        u = Queue.pop()
        for v in G.adj[u]:
            if dist[v] > dist[u] + G.w(u, v):
                dist[v] = dist[u] + G.w(u, v)
                Queue.push(v)
    return dist
```

16) Shortest Path in Graph with Negative Weight

```
Shortest_Path(graph G, vertex s):
    for v in G.vertices:
        dist[v] = float('inf')
    dist[s] = 0
    for i in range(len(G.vertices) - 1):
        for u in G.vertices:
            for v in G.adj[u]:
                if dist[v] > dist[u] + G.w(u, v):
                    dist[v] = dist[u] + G.w(u, v)
    for u in G.vertices:
        for v in G.adj[u]:
            if dist[v] > dist[u] + G.w(u, v):
                return False
    return True
```

17) Shortest Path in Graph with Negative Weight (Bellman-Ford)

```
Bellman_Ford(graph G, vertex s):
    for v in G.vertices:
        dist[v] = float('inf')
    dist[s] = 0
    for i in range(len(G.vertices) - 1):
        for u in G.vertices:
            for v in G.adj[u]:
                if dist[v] > dist[u] + G.w(u, v):
                    dist[v] = dist[u] + G.w(u, v)
    for u in G.vertices:
        for v in G.adj[u]:
            if dist[v] > dist[u] + G.w(u, v):
                return False
    return True
```

18) Shortest Path in Graph with Negative Weight (Bellman-Ford)

```
Bellman_Ford(graph G, vertex s):
    for v in G.vertices:
        dist[v] = float('inf')
    dist[s] = 0
    for i in range(len(G.vertices) - 1):
        for u in G.vertices:
            for v in G.adj[u]:
                if dist[v] > dist[u] + G.w(u, v):
                    dist[v] = dist[u] + G.w(u, v)
    for u in G.vertices:
        for v in G.adj[u]:
            if dist[v] > dist[u] + G.w(u, v):
                return False
    return True
```

19) Shortest Path in Graph with Negative Weight (Bellman-Ford)

```
Bellman_Ford(graph G, vertex s):
    for v in G.vertices:
        dist[v] = float('inf')
    dist[s] = 0
    for i in range(len(G.vertices) - 1):
        for u in G.vertices:
            for v in G.adj[u]:
                if dist[v] > dist[u] + G.w(u, v):
                    dist[v] = dist[u] + G.w(u, v)
    for u in G.vertices:
        for v in G.adj[u]:
            if dist[v] > dist[u] + G.w(u, v):
                return False
    return True
```

20) Shortest Path in Graph with Negative Weight (Bellman-Ford)

```
Bellman_Ford(graph G, vertex s):
    for v in G.vertices:
        dist[v] = float('inf')
    dist[s] = 0
    for i in range(len(G.vertices) - 1):
        for u in G.vertices:
            for v in G.adj[u]:
                if dist[v] > dist[u] + G.w(u, v):
                    dist[v] = dist[u] + G.w(u, v)
    for u in G.vertices:
        for v in G.adj[u]:
            if dist[v] > dist[u] + G.w(u, v):
                return False
    return True
```