GRAPHS

You can represent a graph like G=(V,E,c) ➔

E = Set of edges
V = Set of vertices = nodes
c = weight

|E| = Size of a graph: the number of edges in the graph
|V| = Order of a graph: the number of nodes in the graph
deg(v) = Degree of a vertex: the number of edges adjacent to v.

For a directed graph we have e=(u,v) ➔ u = start vertex, v = end vertex
deg^-(v) = In-degree: the number of edges coming into v in a directed graph.
deg^+(v) = Out-degree: the number of edges going out of v in a directed graph.

Adjacency matrix: (Adj) is a nxn (n = |V|) matrix which represents the edges between vertices.

$$Adj(i, j) = \begin{cases} 1 \text{ if the vi and vj are adjacent} \\ 0 \text{ otherwise} \end{cases}$$

Breadth-First Search (BFS)

Time complexity: O(V+E) if we use adjacency lists

You can use BFS to/for:
1. Start from an initial node (u) to see what vertices are reachable from u.
2. Calculate the shortest distance (**d(u,v)**) from node (u) to any other node (v) in an unweighted graph.
3. Garbage collection
4. Social networking
5. Network broadcast
6. Web crawling
7. Solve puzzles and games like rubik's cube.
8. Test if a graph bipartite (its vertices can be divided into two disjoint sets (no edge between the vertices in one set)

BFS(s):

```
    Initialize FIFO queue Q as being empty.
    Q.push(s)
    s.parent  =  -1
    while Q != null:
        u = Q.pop()                 // Remove node u from Q.
        for v ϵ u.adj               // for every neighbor of u:
            if v.parent == null
                v.parent = u
                Q.push(v)
```

Depth First Search (DFS)

Time complexity: O(V+E) if we use adjacency lists

You take a path and keep going until you reach a dead-end, then you go back and take another path till you explore all the possible paths/graph. The algorithm is exactly the same as BFS, the only difference is that it uses a **stack** instead of a FIFO queue.

You can use DFS to/for:

1. Start from an initial node (u) to see what vertices are reachable from u.
2. Edge classification
3. Cycle detection
4. Topological sorting: job scheduling
5. Solve mazes.

```
timer = 0;
DFS(V)                          // Explore the whole graph
        for s ϵ V
                if s.parent == null
                        s.parent = -1
                        DFS_visit(s)



DFS_visit(s):                   // Start from a node (u) to see what vertices are reachable from that
        timer++
        s.start = timer
        for v ϵ s.adj
                if v.start  == null          // or if v.parent == null
                        v.parent = s
                        DFS_visit(v)
                else if v.start != null && v.end == null
                        // Cycle detected!
                end
        timer++
        s.end = timer
```

Edge classification:

Tree edge: the edge we take to visit a new vertex ((x,y)ϵE ➔ x.start < y.start < y.end < x.end)
Forward edge: connects a node to a descendant ((x,y)ϵE ➔ x.start < y.start < y.end < x.end)
Backward edge: connects a node to an ancestor ((x,y)ϵE ➔ y.start< x.start < x.end <y.end)
Cross edge: all the other edges ((x,y)ϵE ➔ y.start < y.finish < x.start < x.finish)

Cycle detection:

- ✓ G has e cycle iff G has a back edge.

Topological sorting: gives you a topological order/sort of vertices of a **directed acyclic graph** (DAG). For every edge in a DAG ((v_i ,v_j)ϵ E) we have v_i is before v_j in a topological order of v_1, v_2, …, v_n.

- ✓ Run DFS to calculate the start and end time of the vertices. (Make sure that you have a DAG before going to the next step).

- ✓ As each vertex finishes, insert it onto a linked list.
- ✓ Printing the linked list, gives you the topological order of the vertices.

Dijkstra Algorithm

Time complexity: O((E+V)logV) if we use a min-heap

This algorithm calculates the shortest distance (**d(u,v)**) from node (u) to any other node (v).

```
Dijkstra(V, s)
    for v ϵ V
        v.dst = inf;
    s.dst = 0;
    s.parent = -1
    min_heap.build(V)                       // Build a min-heap based on dst
    while min_heap ≠ null:
        u = min_heap.delete(root)           // Choose a node with the smallest distance.
        for v ϵ u.adj                       // for every neighbor of u
            if v.parent == null             // Not explored
                v.parent = u
            if u.dst + w(u,v)< v.dst
                v.parent = u.
                v.dst = u.dst + w(u,v)
                min_heap.heapify(v)         // Update the min_heap
```

Minimum Spanning Tree (MST)

Tree: a tree with **n** nodes has **n-1** edges. ➔ **|E|=|V|-1** and does not have any cycles.

Spanning tree: a tree that is connected to all the nodes in a given connected undirected graph.

Minimum Spanning Tree (MST): a spanning tree with the smallest weight between all the other spanning trees.

Prim's Algorithm

Time complexity: O($ElogV$) if we use min-heaps

This algorithm finds MST on a given connected undirected graph.

1. Initialize an empty tree
2. Choose a random node and add it to your tree.

3. Find an edge with the smallest weight among the edges that connect a vertex of the tree to a vertex **not** in the tree.
4. Add the selected edge and its vertex to the tree.
5. Go back to step 3 until you cover all the vertices in your tree (|E|=|V|-1)

```
Prim(V, s)
        for v ϵ V
                v.cost = inf;
        s.cost = 0;
        s.parent = -1
        min_heap.build(V)                          // Build a min-heap based on cost
        while min_heap ≠ null:
                u = min_heap.delete(root)          // Choose a node with the smallest cost.
                for v ϵ u.adj                      // for every neighbor of u
                        if v.parent == null        // Not explored
                                v.parent = u
                        if w(u,v) < v.cost
                                v.parent = u.
                                v.cost = w(u,v)
                                min_heap.heapify(v)  // Update the min_heap
```

Kruskal's Algorithm

Time complexity: O($ElogV$)

Kruskal's Algorithm also finds MST on a given connected undirected graph.

1. Sort the edges based on their weights
2. Picking an edge with the smallest weight **if it does not make a cycle with the chosen ones**
3. Go back to Step 2 until the number of the chosen edges become |V|-1 This algorithm finds MST on a given connected undirected graph.