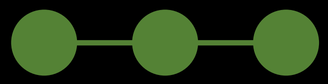
# Graphs

Data Structures & Algorithm

# Categories of Data Structures

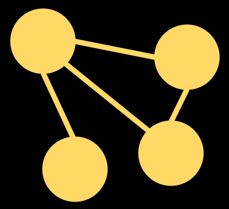
**Linear Ordered**

- Lists
- Stacks
- Queues



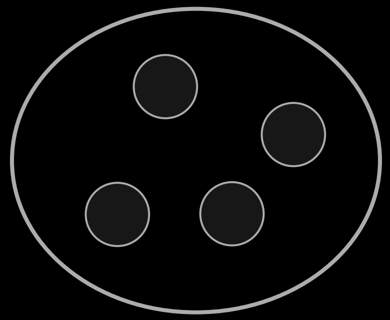
**Non-linear Ordered**

- Trees
- Graphs



**Not Ordered**

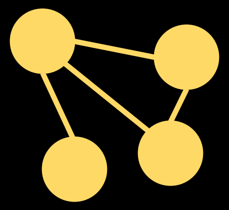
- Sets
- Tables/Maps



# Recap

Non-linear Ordered

Graphs

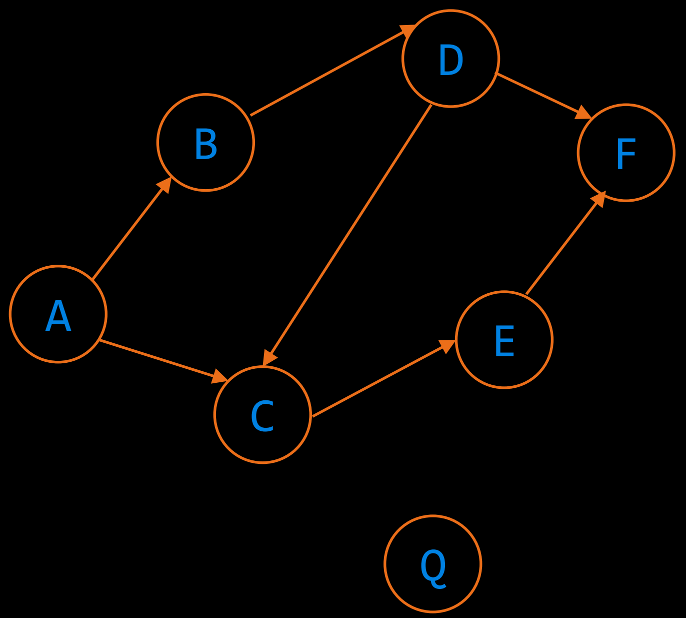


- **Graphs**
  - Terminology
  - Types
- **Graph Implementations**
  - Edge List
  - Adjacency Matrix
  - Adjacency List

# Shortest Path

# s-t Path

**Is there a path between vertices s and t?**

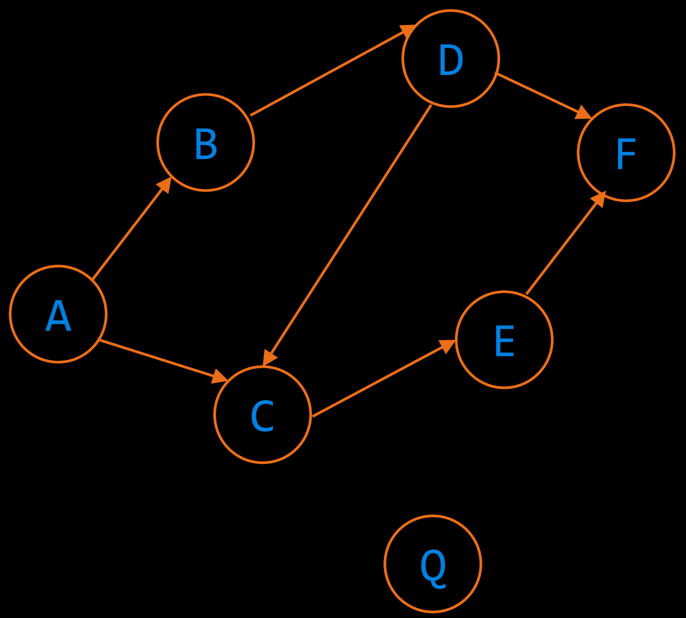



**Is there a path between vertices A and C? - Yes**

**Is there a path between vertices A and Q? - No**

# s-t Path

## Is there a path between vertices s and t?



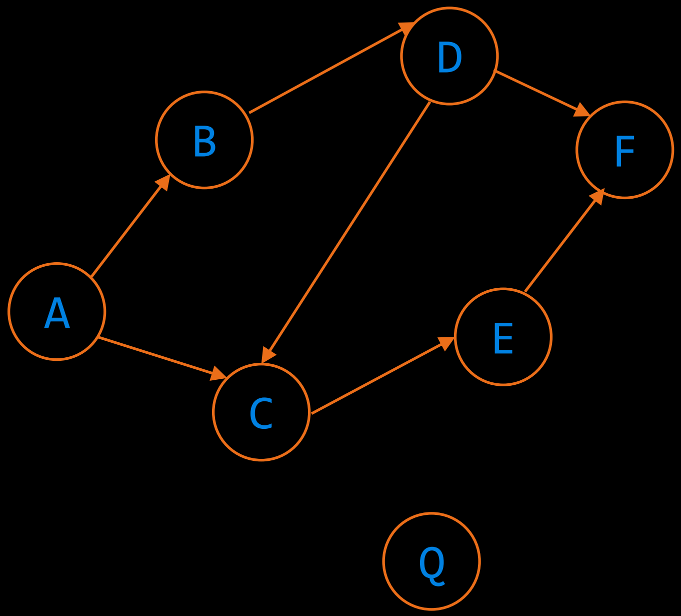

Is there a path between vertices A and C? - Yes

Is there a path between vertices A and Q? - No

**Solution**

Perform DFS or BFS with source "s" and if we encounter "t" in the path/traversal, then return True otherwise False

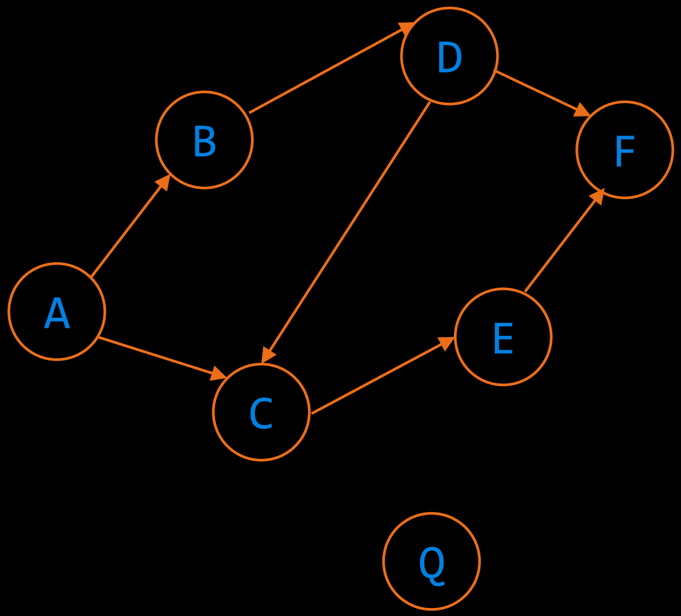
# 7.2.1 DFS to Find Whether a Given Vertex is Reachable (Iterative)




s-t Path

```
bool dfs(const Graph& graph, int src, int dest)
{
    set<int> visited;
    stack<int> s;
    visited.insert(src);
    s.push(src);
    while(!s.empty())
    {
        int u = s.top();
        s.pop();
        for(auto v: graph.adjList[u])
        {
            if(v == dest)
                return true;
            if ((visited.find(v) == visited.end()))
            {
                visited.insert(v);
                s.push(v);
            }
        }
    }
    return false;
}
```

# 7.2.1 DFS to Find Whether a Given Vertex is Reachable (Recursive)



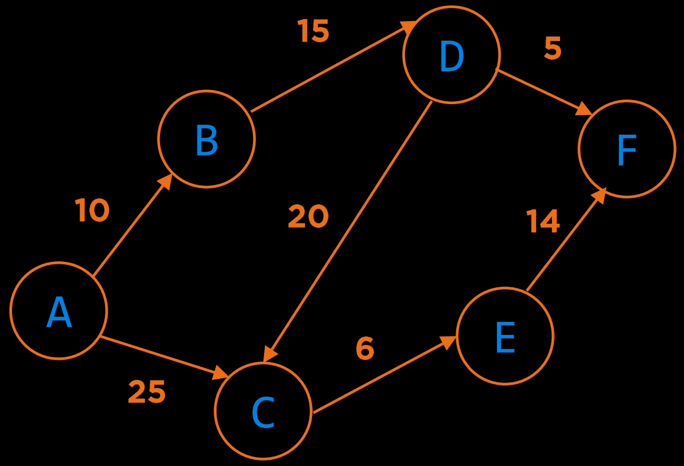

s-t Path: Recursive

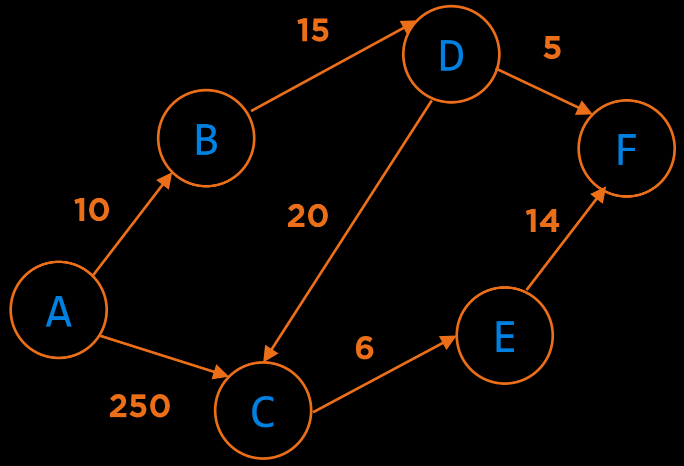
```
bool dfs_helper(const Graph& graph, int src, int dest, vector<bool>& visited)
{
    visited[src] = true;

    if (src == dest)
        return true;

    for (int neighbor : graph.adjList[src]) {
        if (!visited[neighbor]) {
            if (dfs_helper(graph, neighbor, dest, visited))
                return true;
        }
    }
    return false;
}

bool dfs(const Graph& graph, int src, int dest)
{
    vector<bool> visited(graph.numVertices);
    return dfs_helper(graph, src, dest, visited);
}
```

# Problem with s-t Path

**What if the edges are weighted?**

**The algorithms do not consider the weights.**



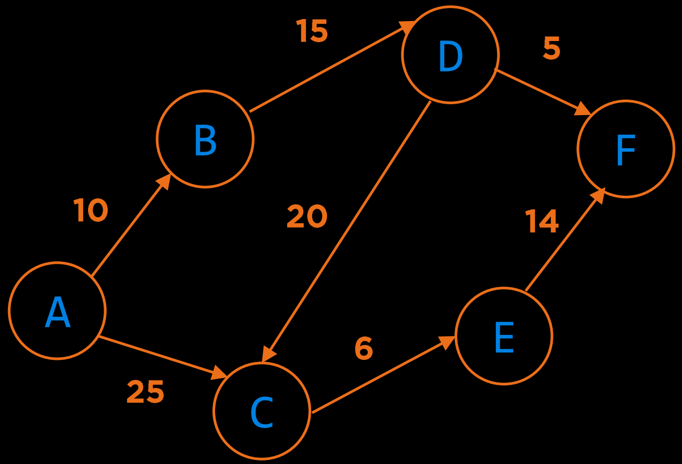



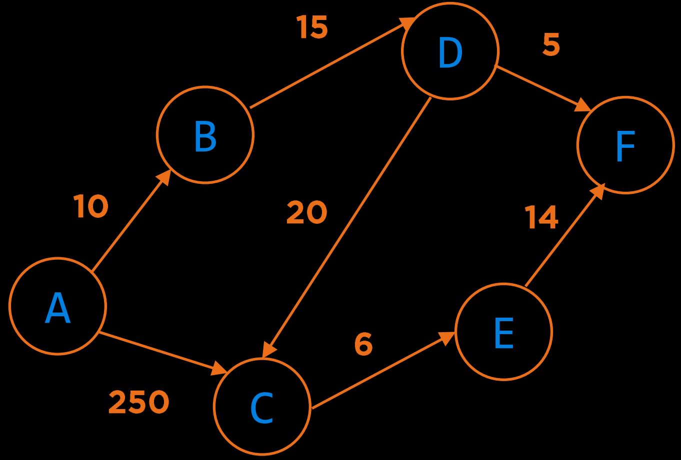
# Problem with s-t Path

What if the edges are weighted?

The algorithms do not consider the weights.

**Example 1:** Path for A to C will be A-B-D-C for a DFS traversal which will have a total cost of 45 against 25 for the path directly from A-C.

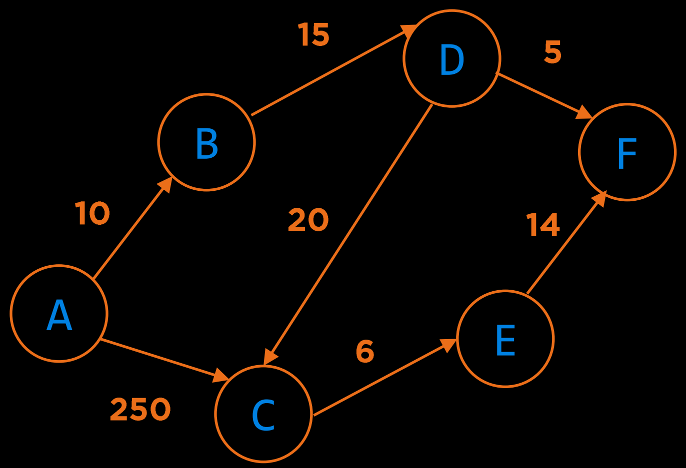



**Example 2:** Path for A to C will be A-C for a BFS traversal which might have a total cost of 250 against 45 for the path directly from A-B-D-C.

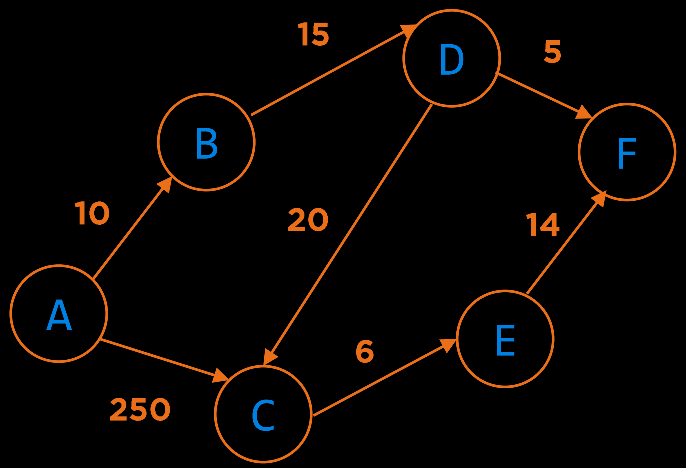
# Shortest Weighted s-t Path

What is the shortest weighted path between vertices s and t?



# Shortest Weighted s-t Path

**What is the shortest weighted path between vertices s and t?**

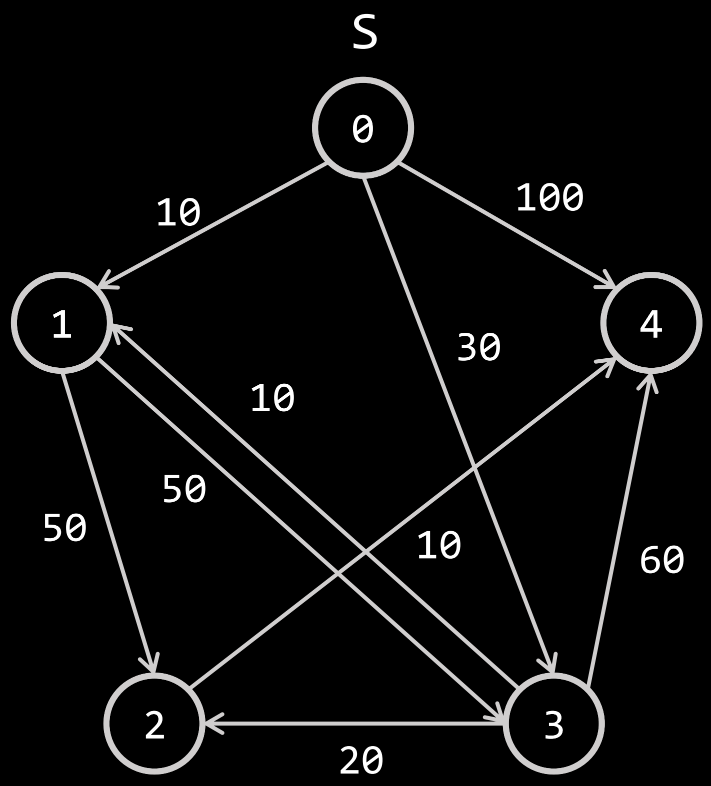



- Dijkstra's Algorithm
  - Single Source: Path to all vertices
  - Directed Graphs
  - No negative weights allowed
  - No negative weight cycles allowed
- Bellman Ford
  - Single Source: Path to all vertices
  - Negative Weights allowed
  - No negative weight cycles allowed
- Floyd-Warshall
  - All pair shortest paths
- A* Search

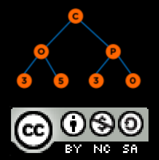
# Dijkstra's Shortest Path Algorithm

## Example

- Specify a source vertex, S
- Initialize two arrays and two sets
    - Set S will contain the vertices for which we have computed the shortest distance
        - Initially S will be empty
    - Set V-S will contain the vertices we still need to process
        - Initialize V-S by placing all vertices into it
    - d[v] will contain shortest distance from s to v
        - Initially all d[v]'s will be set to infinity except for source which will be 0
    - p[v] will predecessor of v in the path from s to v
        - Initially all p[v]'s will be set to -1





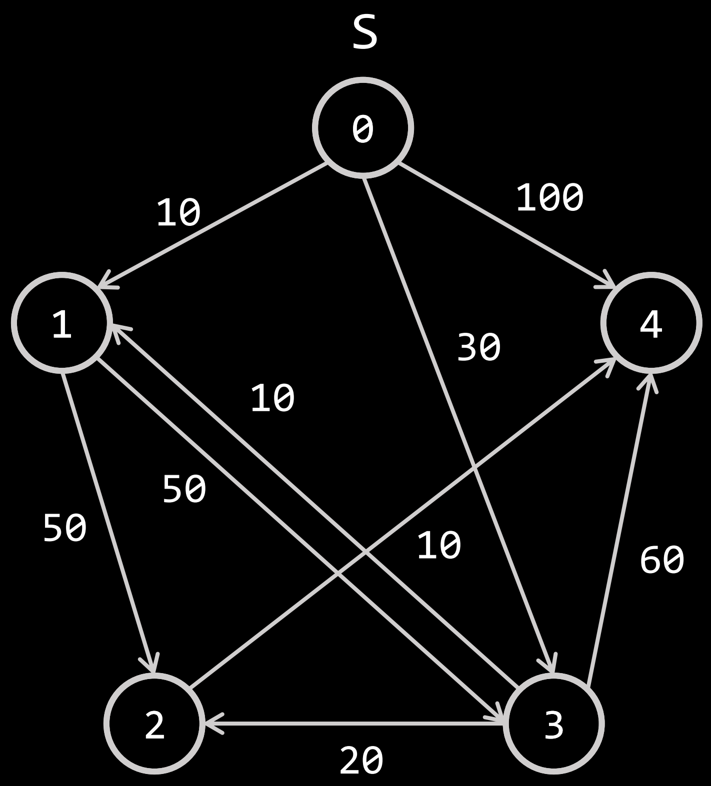
# Dijkstra's Shortest Path Algorithm

## Example

Computed, S = {}

Needs processing, V-S = {0, 1, 2, 3, 4}

| v | d[v] | p[v] |
|---|---|---|
| 0 | 0 | -1 |
| 1 | ∞ | -1 |
| 2 | ∞ | -1 |
| 3 | ∞ | -1 |
| 4 | ∞ | -1 |



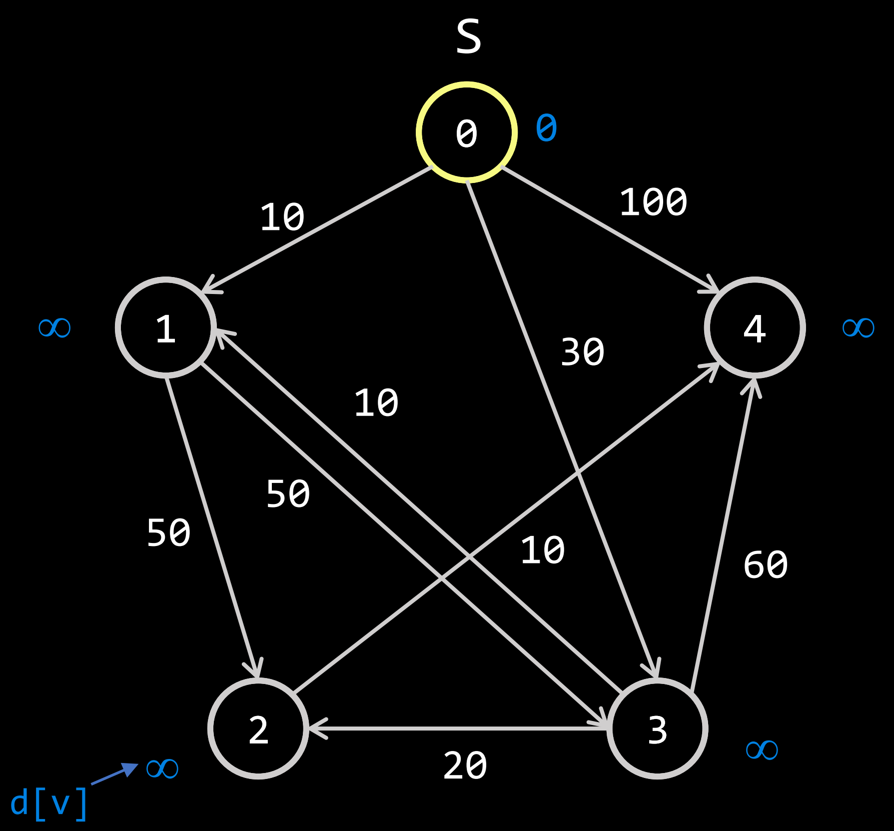
# Dijkstra's Shortest Path Algorithm

**Example: Start with vertex that has minimum distance in d[v], i.e. 0 and add to Computed**

Computed, S = {0}

Needs processing, V-S = {1, 2, 3, 4}

| v | d[v] | p[v] |
|---|---|---|
| 0 | 0 | -1 |
| 1 | ∞ | -1 |
| 2 | ∞ | -1 |
| 3 | ∞ | -1 |
| 4 | ∞ | -1 |



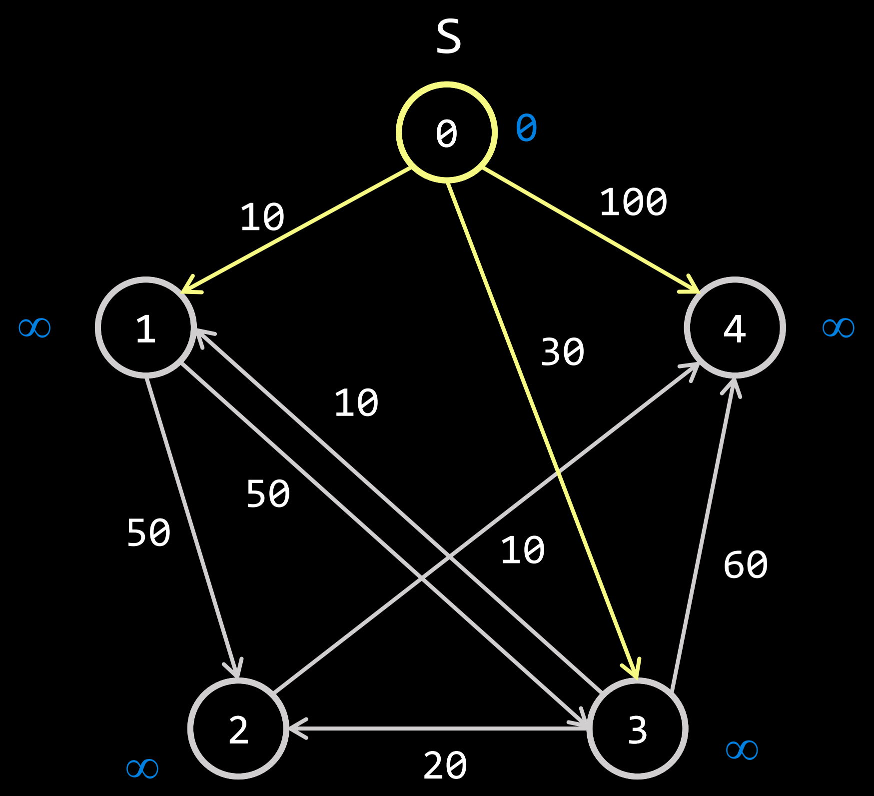
# Dijkstra's Shortest Path Algorithm

**Example: Process edges adjacent to the vertex 0 and update distances based on relaxation***

Computed, S = {0}

Needs processing, V-S = {1, 2, 3, 4}

| v | d[v] | p[v] |
|---|---|---|
| 0 | 0 | -1 |
| 1 | ∞ | -1 |
| 2 | ∞ | -1 |
| 3 | ∞ | -1 |
| 4 | ∞ | -1 |




```
* Relaxation
  if (dist[v] > dist[u] + w)
          dist[v] = dist[u] + w;
```

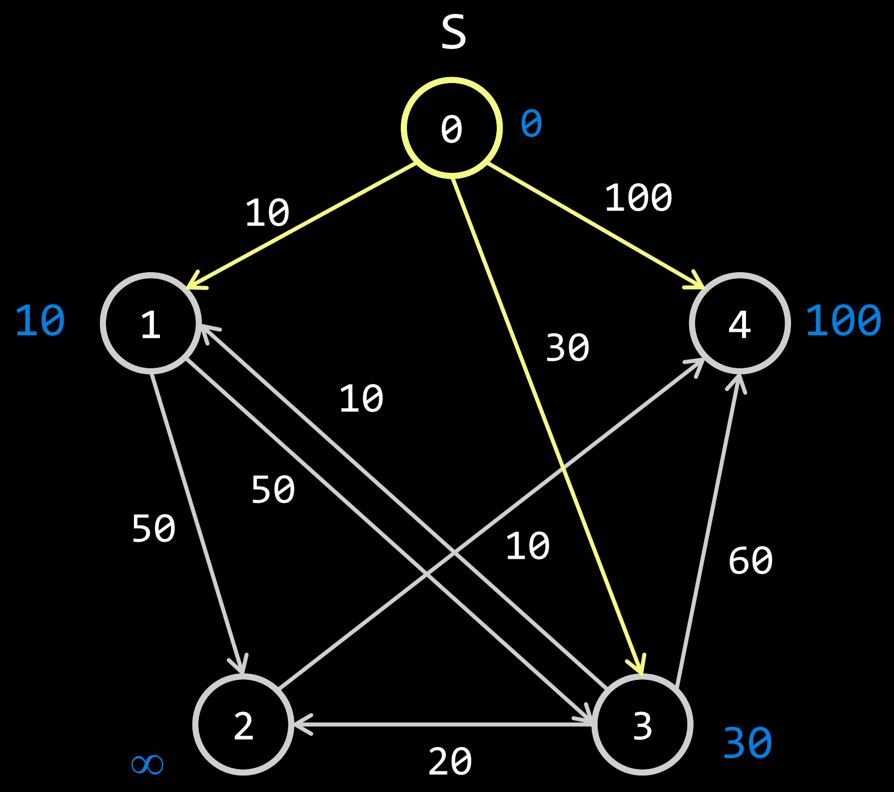
# Dijkstra's Shortest Path Algorithm

**Example: Process edges adjacent to the vertex 0 and update distances based on relaxation***

Computed, S = {0}

Needs processing, V-S = {1, 2, 3, 4}

| v | d[v] | p[v] |
|---|---|---|
| 0 | 0 | -1 |
| 1 | 10 | 0 |
| 2 | ∞ | -1 |
| 3 | 30 | 0 |
| 4 | 100 | 0 |




* Relaxation

```
if (dist[v] > dist[u] + w)
        dist[v] = dist[u] + w;
```

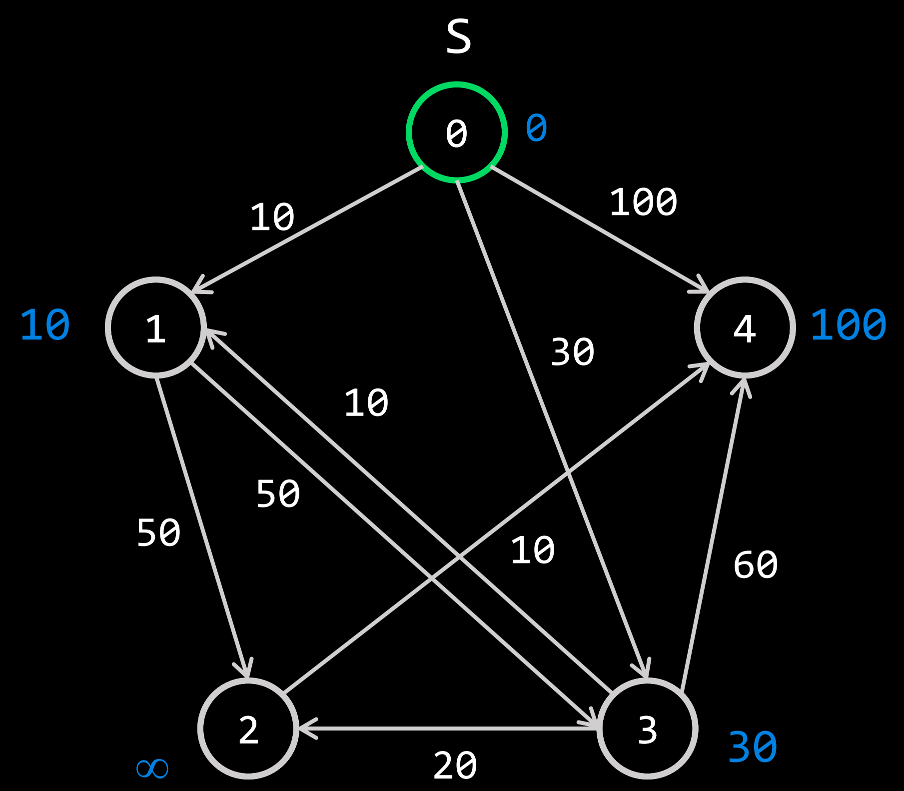
# Dijkstra's Shortest Path Algorithm

**Example: 0 is now done. Next, repeat the process picking the minimum element in d[v] that has not been computed**

Computed, S = {0}

Needs processing, V-S = {1, 2, 3, 4}

| v | d[v] | p[v] |
|---|---|---|
| 0 | 0 | -1 |
| 1 | 10 | 0 |
| 2 | ∞ | -1 |
| 3 | 30 | 0 |
| 4 | 100 | 0 |



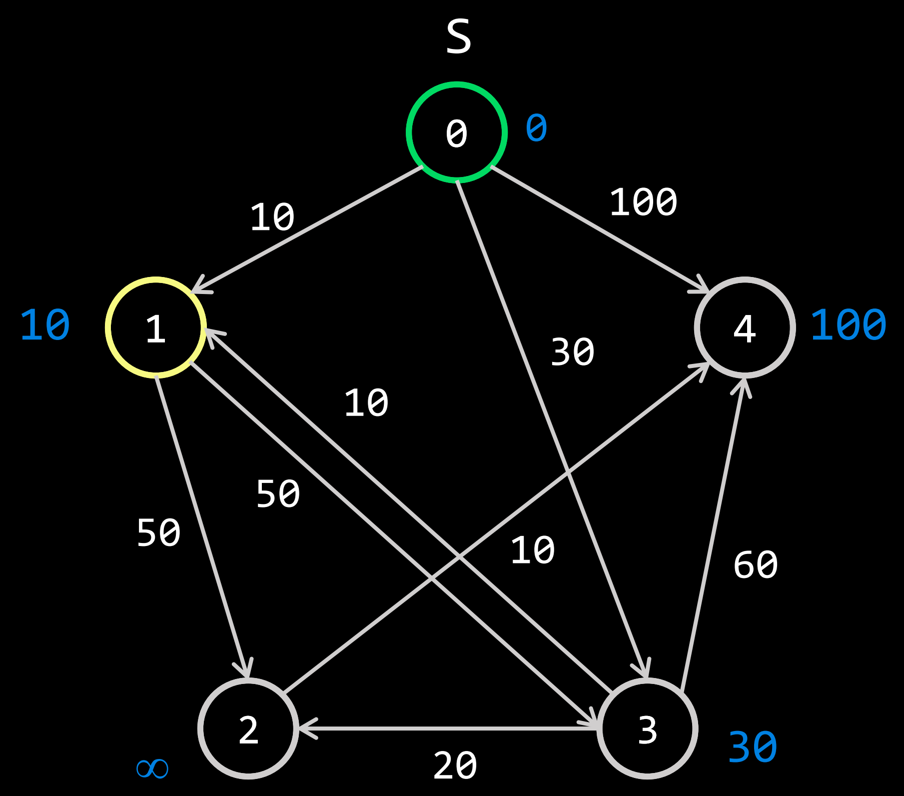
# Dijkstra’s Shortest Path Algorithm

**Example: Pick 1**

Computed, S = {0}

Needs processing, V-S = {1, 2, 3, 4}

| v | d[v] | p[v] |
|---|---|---|
| 0 | 0 | -1 |
| 1 | 10 | 0 |
| 2 | ∞ | -1 |
| 3 | 30 | 0 |
| 4 | 100 | 0 |



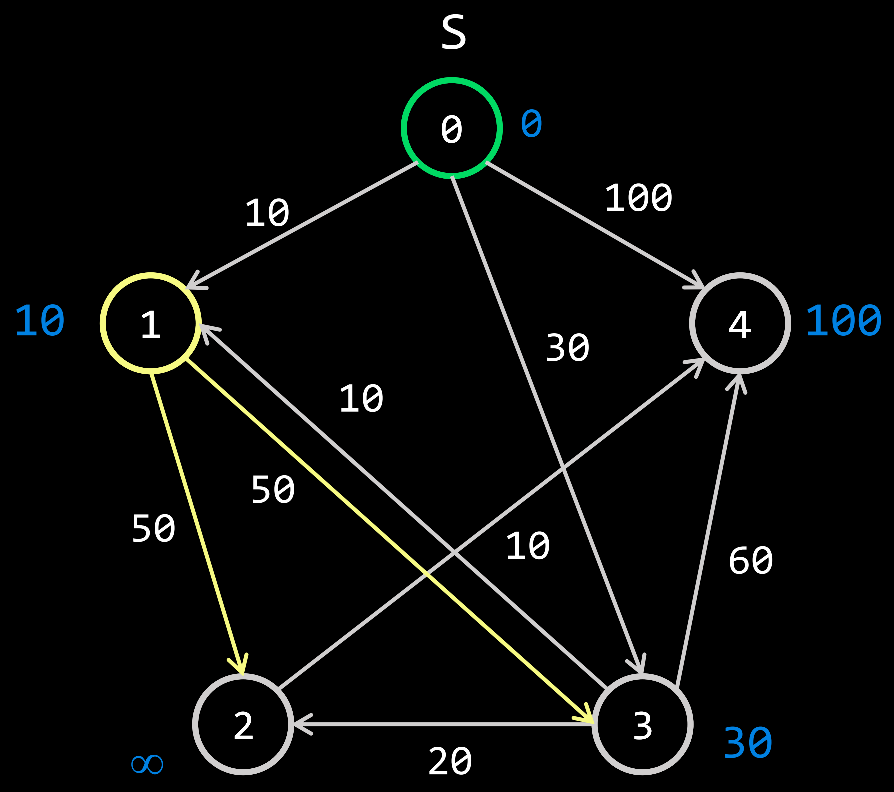
# Dijkstra's Shortest Path Algorithm

**Example: Process edges adjacent to the vertex 1 and update distances based on relaxation**

Computed, S = {0}

Needs processing, V-S = {1, 2, 3, 4}

| v | d[v] | p[v] |
|---|---|---|
| 0 | 0 | -1 |
| 1 | 10 | 0 |
| 2 | ∞ | -1 |
| 3 | 30 | 0 |
| 4 | 100 | 0 |



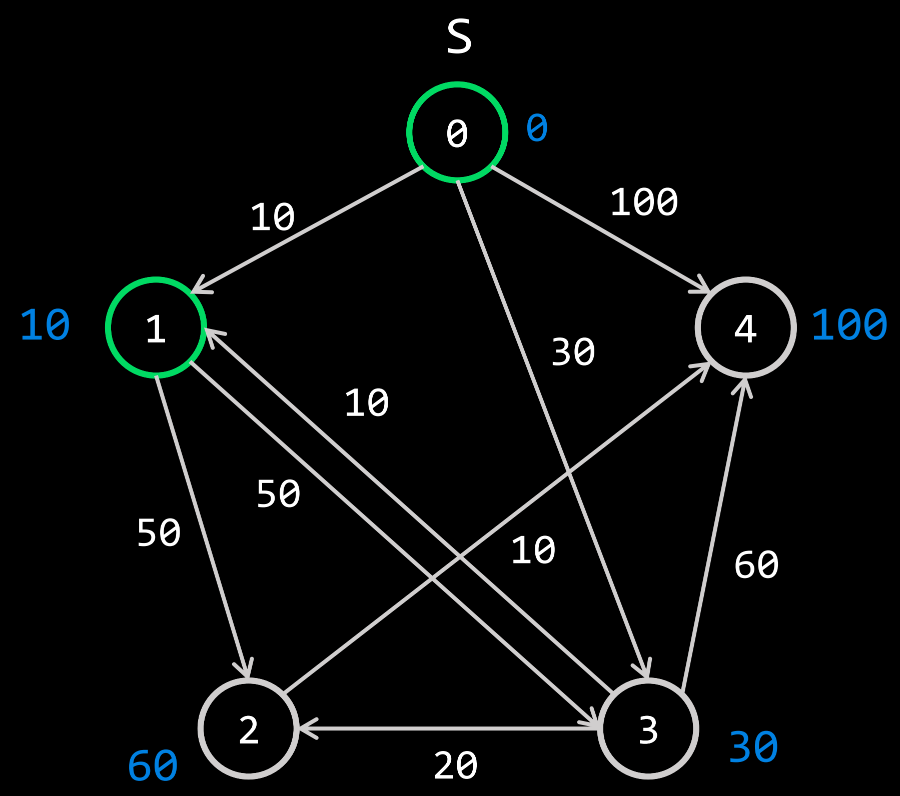
# Dijkstra's Shortest Path Algorithm

**Example: 1 is now done. Next, repeat the process picking the minimum element in d[v] that has not been computed**

Computed, S = {0, 1}

Needs processing, V-S = {2, 3, 4}

| v | d[v] | p[v] |
|---|---|---|
| 0 | 0 | -1 |
| 1 | 10 | 0 |
| 2 | 60 | 1 |
| 3 | 30 | 0 |
| 4 | 100 | 0 |



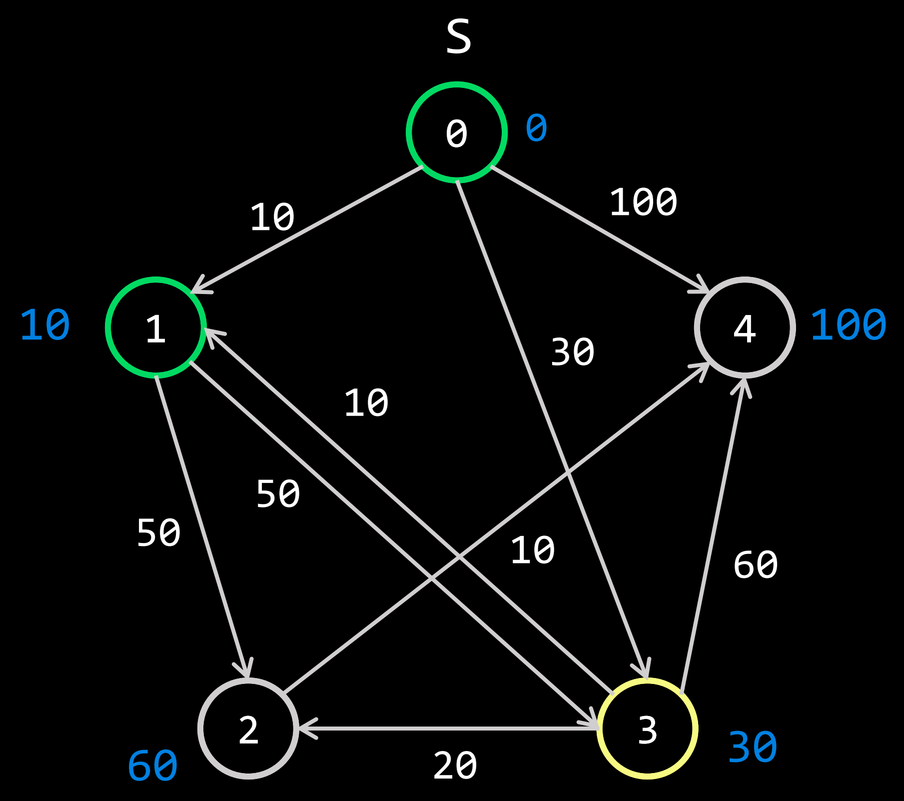
# Dijkstra's Shortest Path Algorithm

**Example: Pick 3**

Computed, S = {0, 1}

Needs processing, V-S = {2, 3, 4}

| v | d[v] | p[v] |
|---|---|---|
| 0 | 0 | -1 |
| 1 | 10 | 0 |
| 2 | 60 | 1 |
| 3 | 30 | 0 |
| 4 | 100 | 0 |



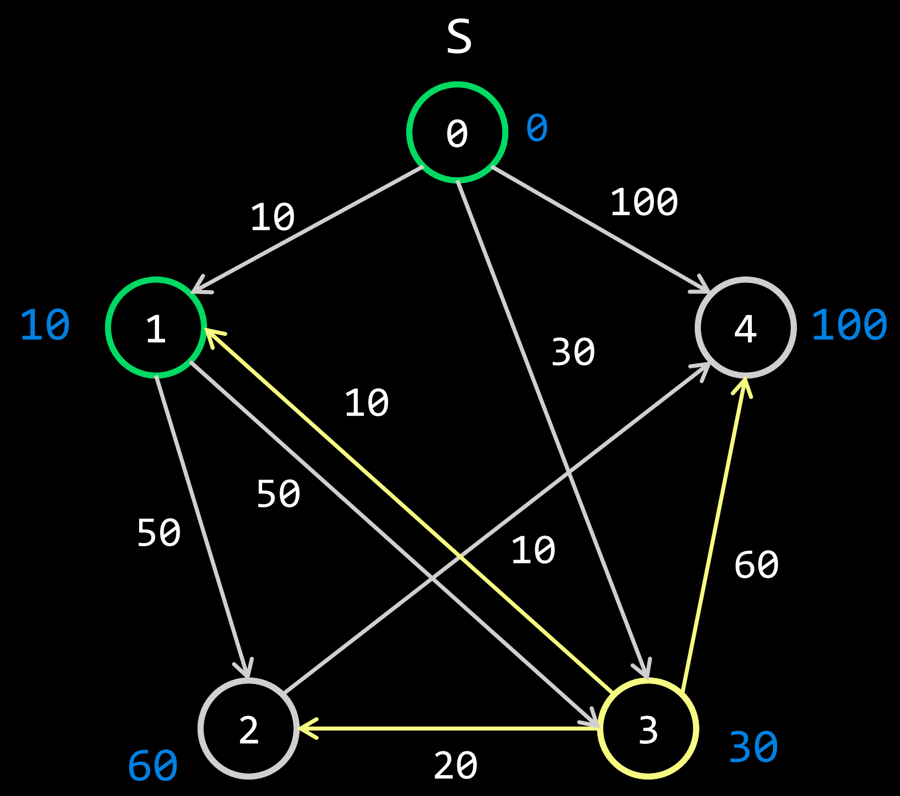
# Dijkstra's Shortest Path Algorithm

**Example: Process edges adjacent to the vertex 3 and update distances based on relaxation**

Computed, S = {0, 1, 3}

Needs processing, V-S = {2, 4}

| v | d[v] | p[v] |
|---|---|---|
| 0 | 0 | -1 |
| 1 | 10 | 0 |
| 2 | 60 | 1 |
| 3 | 30 | 0 |
| 4 | 100 | 0 |



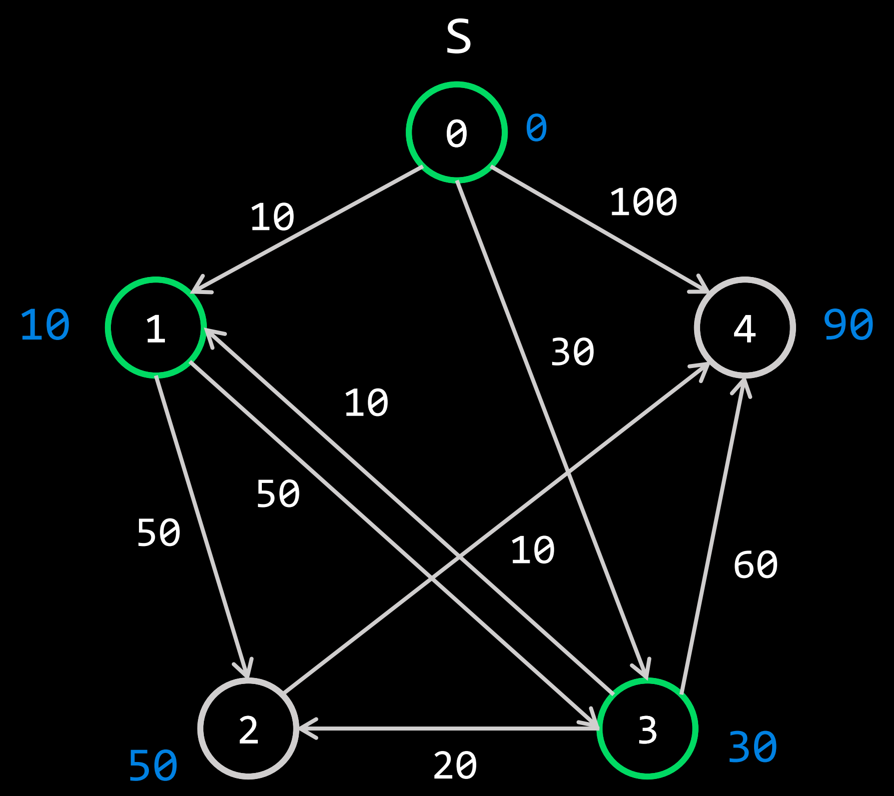
# Dijkstra's Shortest Path Algorithm

**Example: 3 is now done**

Computed, S = {0, 1, 3}

Needs processing, V-S = {2, 4}

| v | d[v] | p[v] |
|---|---|---|
| 0 | 0 | -1 |
| 1 | 10 | 0 |
| 2 | 50 | 3 |
| 3 | 30 | 0 |
| 4 | 90 | 3 |



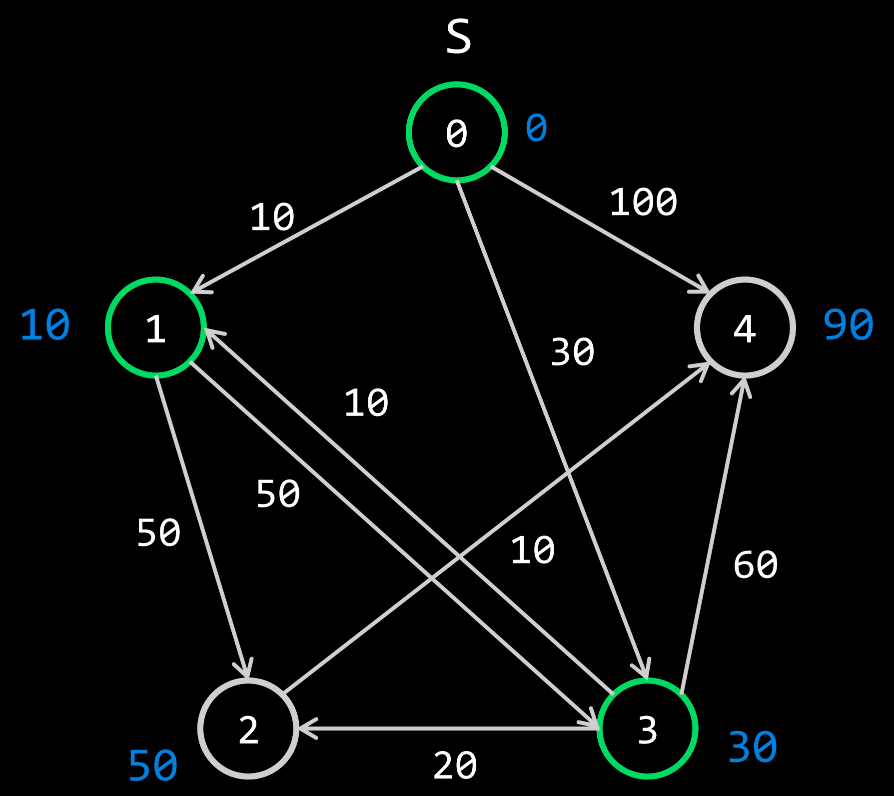
# Dijkstra's Shortest Path Algorithm

**Example: Next, repeat the process picking the minimum element in d[v] that has not been computed**

Computed, S = {0, 1, 3}

Needs processing, V-S = {2, 4}

| v | d[v] | p[v] |
|---|---|---|
| 0 | 0 | -1 |
| 1 | 10 | 0 |
| 2 | 50 | 3 |
| 3 | 30 | 0 |
| 4 | 90 | 3 |



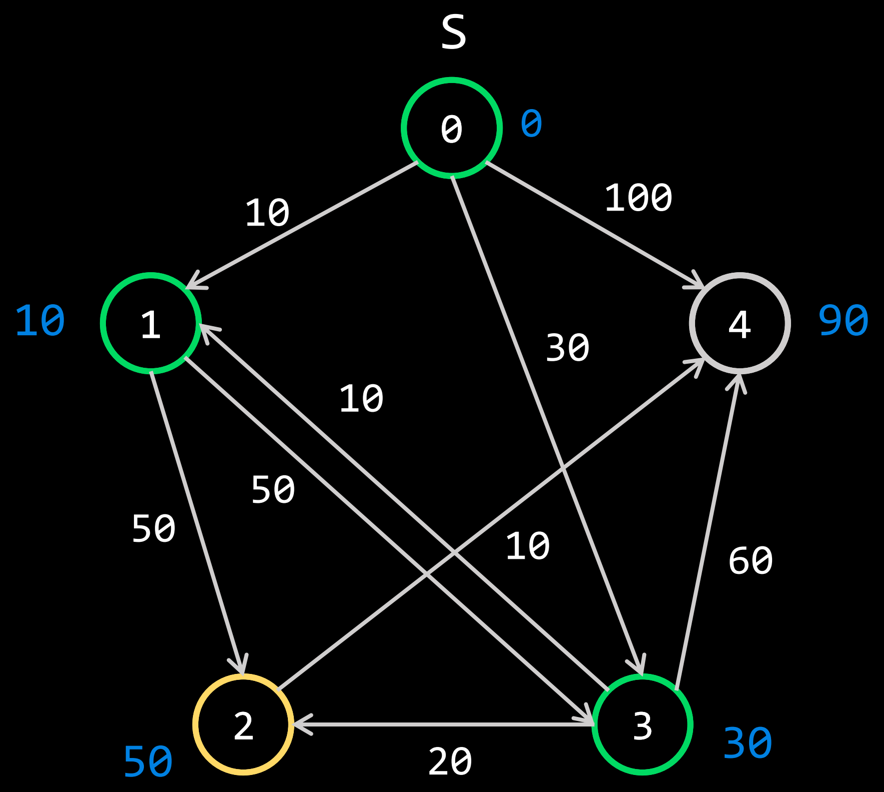
# Dijkstra's Shortest Path Algorithm

**Example: Pick 2**

Computed, S = {0, 1, 3}

Needs processing, V-S = {2, 4}

| v | d[v] | p[v] |
|---|---|---|
| 0 | 0 | -1 |
| 1 | 10 | 0 |
| 2 | 50 | 3 |
| 3 | 30 | 0 |
| 4 | 90 | 3 |



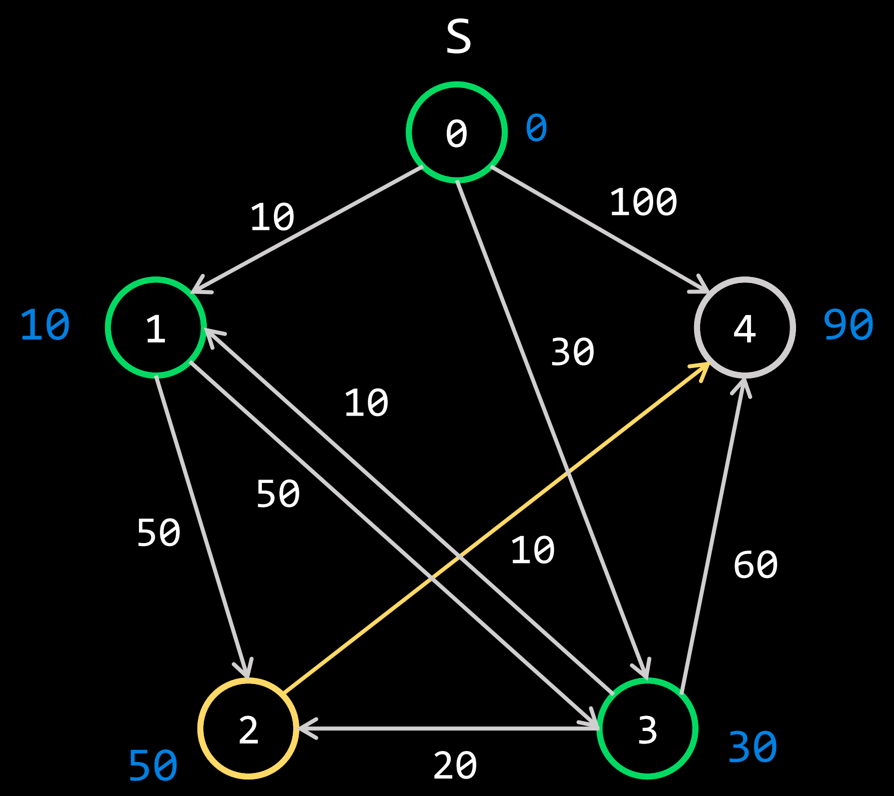
# Dijkstra's Shortest Path Algorithm

**Example: Process edges adjacent to the vertex 2 and update distances based on relaxation**

Computed, S = {0, 1, 2, 3}

Needs processing, V-S = {4}

| v | d[v] | p[v] |
|---|---|---|
| 0 | 0 | -1 |
| 1 | 10 | 0 |
| 2 | 50 | 3 |
| 3 | 30 | 0 |
| 4 | 90 | 3 |



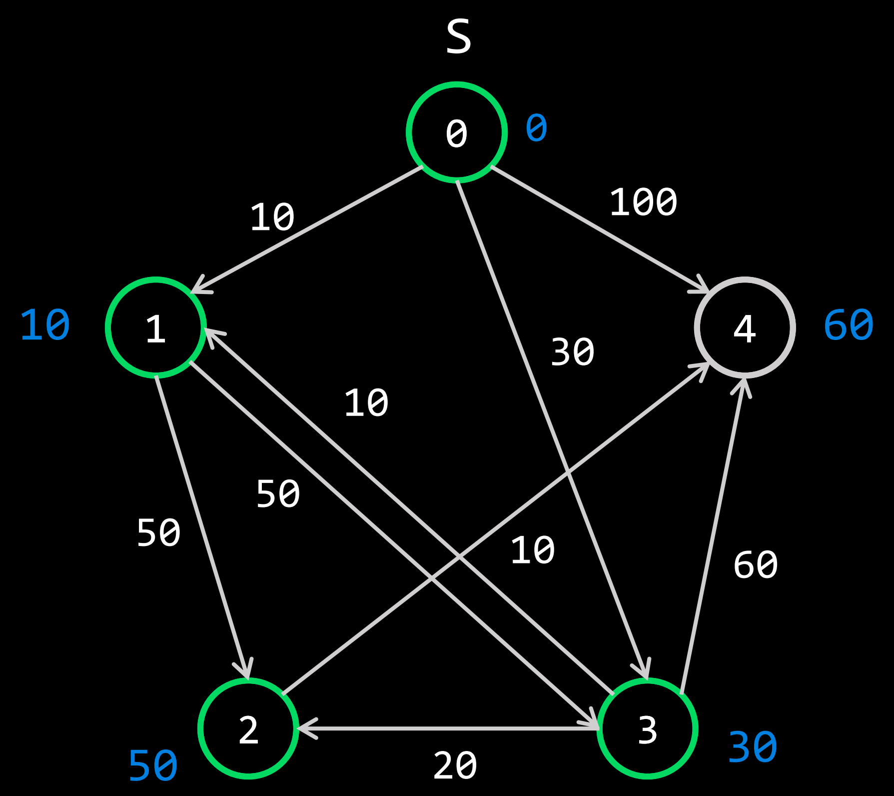
# Dijkstra's Shortest Path Algorithm

**Example: 2 is now done**

Computed, S = {0, 1, 2, 3}

Needs processing, V-S = {4}

| v | d[v] | p[v] |
|---|---|---|
| 0 | 0 | -1 |
| 1 | 10 | 0 |
| 2 | 50 | 3 |
| 3 | 30 | 0 |
| 4 | 60 | 2 |



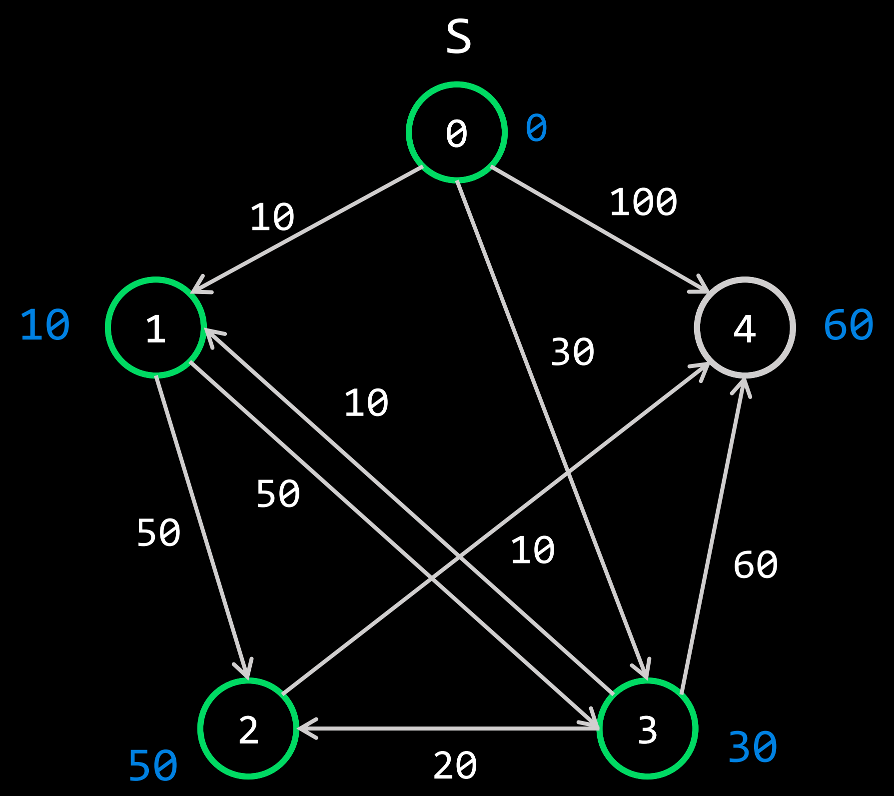
# Dijkstra’s Shortest Path Algorithm

**Example: Next, repeat the process picking the minimum element in d[v] that has not been computed**

Computed, S = {0, 1, 2, 3}

Needs processing, V-S = {4}

| v | d[v] | p[v] |
|---|---|---|
| 0 | 0 | -1 |
| 1 | 10 | 0 |
| 2 | 50 | 3 |
| 3 | 30 | 0 |
| 4 | 60 | 2 |



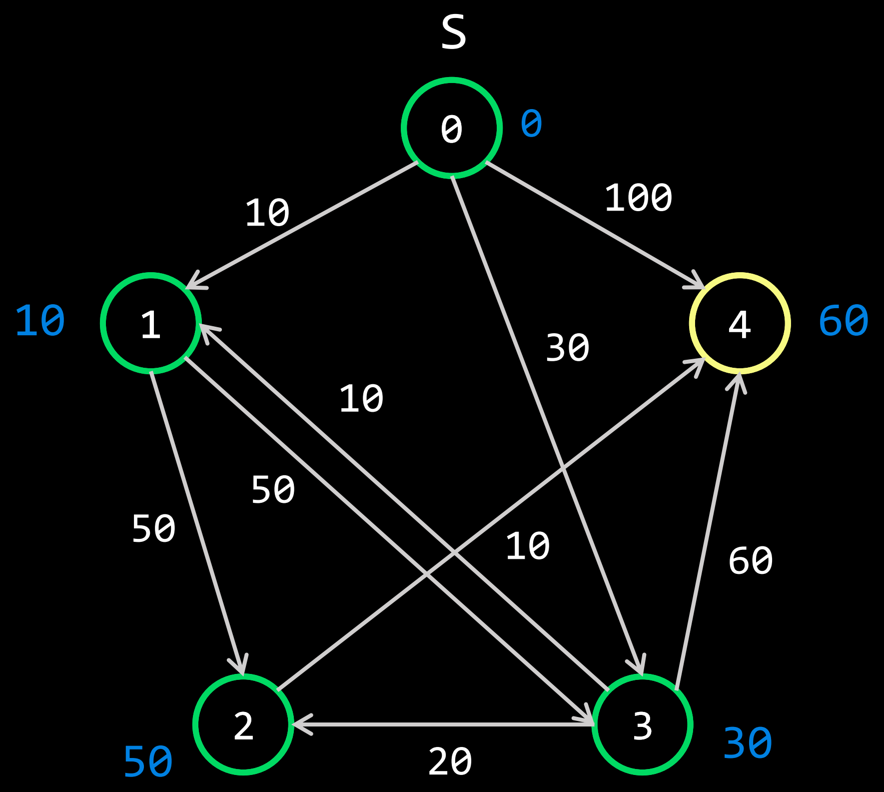
# Dijkstra’s Shortest Path Algorithm

**Example: Pick 4. Process edges adjacent to the vertex 4 and update distances based on relaxation**

Computed, S = {0, 1, 2, 3, 4}

Needs processing, V-S = {}

| v | d[v] | p[v] |
|---|---|---|
| 0 | 0 | -1 |
| 1 | 10 | 0 |
| 2 | 50 | 3 |
| 3 | 30 | 0 |
| 4 | 60 | 2 |



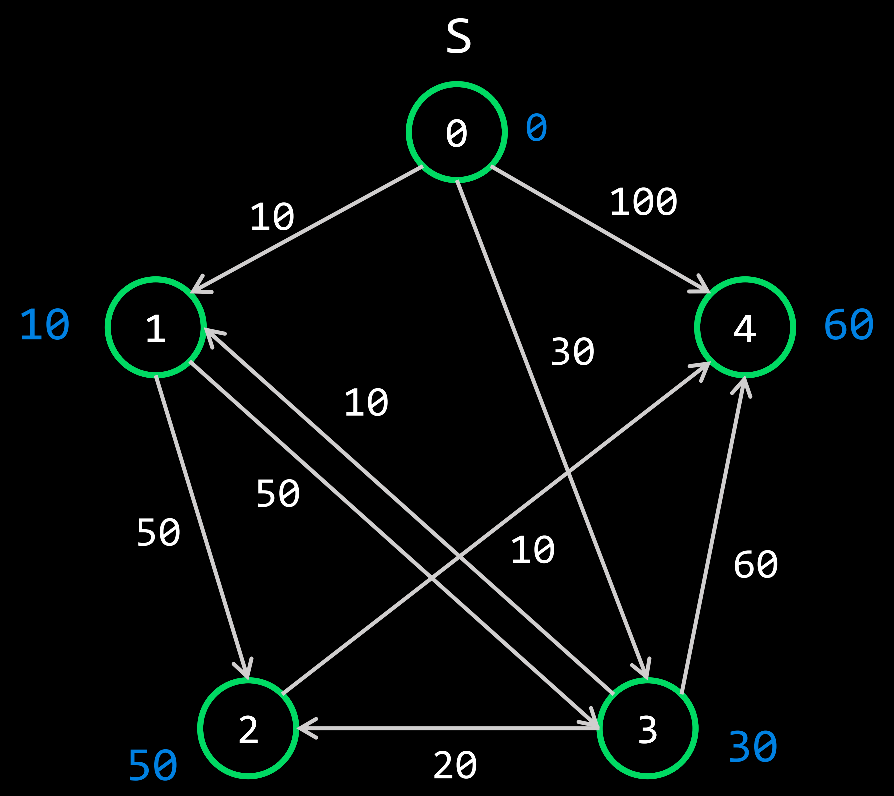
# Dijkstra’s Shortest Path Algorithm

**Example: 4 is now done and V-S is empty. Stop.**

Computed, S = {0, 1, 2, 3, 4}

Needs processing, V-S = {}

| v | d[v] | p[v] |
|---|---|---|
| 0 | 0 | -1 |
| 1 | 10 | 0 |
| 2 | 50 | 3 |
| 3 | 30 | 0 |
| 4 | 60 | 2 |



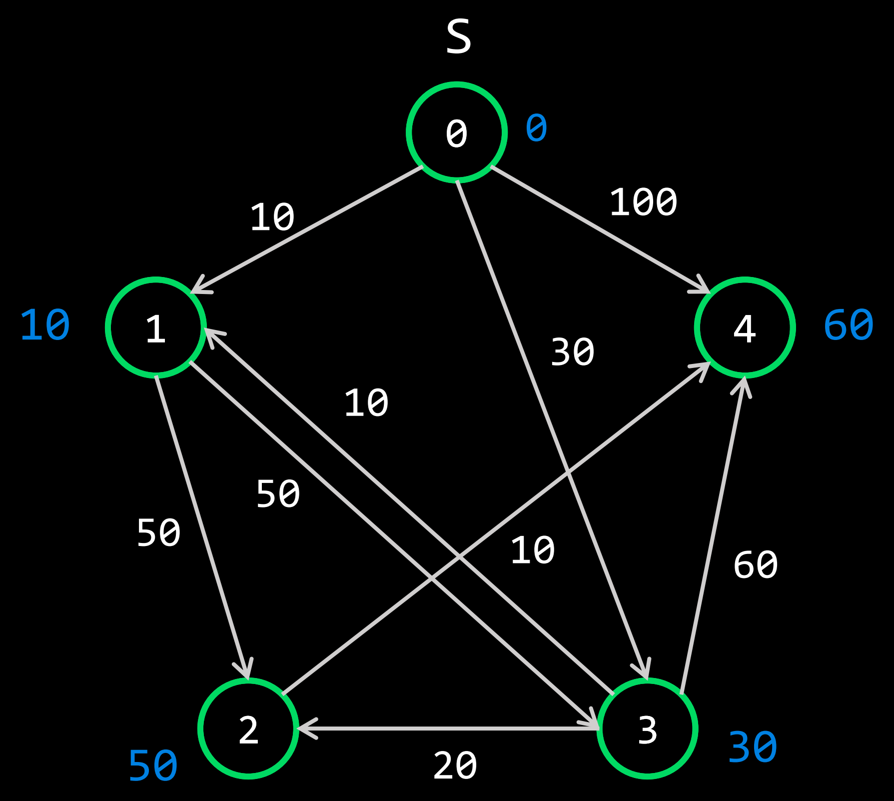
# Dijkstra’s Shortest Path Algorithm

**Example: 4 is now done and V-S is empty. Stop.**

Computed, S = {0, 1, 2, 3, 4}

Needs processing, V-S = {}

| v | d[v] | p[v] |
|---|---|---|
| 0 | 0 | -1 |
| 1 | 10 | 0 |
| 2 | 50 | 3 |
| 3 | 30 | 0 |
| 4 | 60 | 2 |




Path from 0 to 4: 0 3 2 4 (Cost: 60)

# Dijkstra’s Shortest Path Algorithm

**Dijkstra’s Algorithm**

```
1.  Initialize S with the start vertex, s, and V–S with the remaining vertices.
2.  for all v in V–S
3.      Set p[v] to s.
4.      if there is an edge (s, v)
5.          Set d[v] to w(s, v).
        else
6.          Set d[v] to ∞.
7.  while V–S is not empty
8.      for all u in V–S, find the smallest d[u].
9.      Remove u from V–S and add u to S.
10.     for all v adjacent to u in V–S
11.         if d[u] + w(u, v) is less than d[v].
12.             Set d[v] to d[u] + w(u, v).
13.             Set p[v] to u.
```

# Dijkstra's Shortest Path Algorithm

```
Dijkstra's:
    PQ.add(source, 0)
    For other vertices v, PQ.add(v, infinity)
    While PQ is not empty:
        p = PQ.removeSmallest()
        Relax all edges from p

Relaxing an edge u → v with weight w:
    If d[u] + w < d[v]:
        d[v] = d[u] + w
        p[v] = u
        PQ.changePriority(v, d[v])
```

# Dijkstra's Shortest Path Algorithm

```
Dijkstra’s:
    PQ.add(source, 0)                                        O(V*log V)
    For other vertices v, PQ.add(v, infinity)
    While PQ is not empty:
        p = PQ.removeSmallest()                              O(V*log V)
        Relax all edges from p

Relaxing an edge u → v with weight w:
    If d[u] + w < d[v]:
        d[v] = d[u] + w
        p[v] = u
        PQ.changePriority(v, d[v])                           O(E*log V)
```

# Dijkstra's Properties

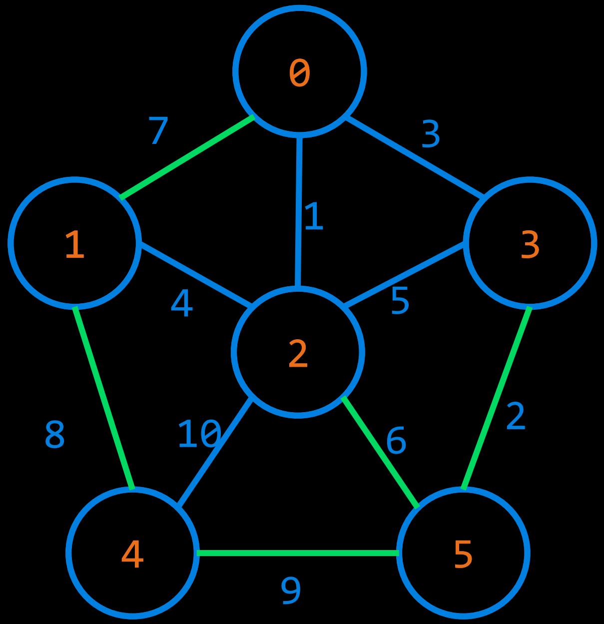
- Greedy Algorithm: Visit vertices in order of best-known distance from source. On visit, relax every edge from the visited vertex
- Dijkstra's is guaranteed to return a correct result if all edges are non-negative.
- Dijkstra's is guaranteed to be optimal so long as there are no negative edges.
- Overall runtime: O(V*log(V) + V*log(V) + E*logV).
  - Assuming E > V, this is just O(E log V) for a connected graph.

# Questions

# Minimum Spanning Tree

# Spanning Tree

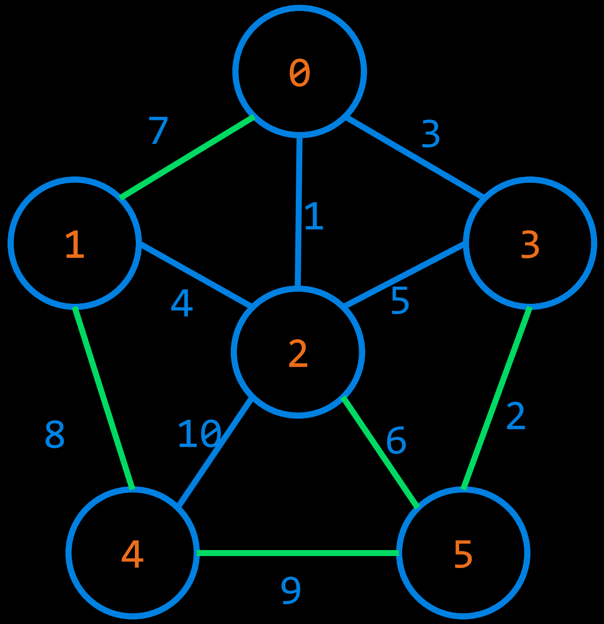
- A spanning tree is a subset of the edges of a graph such that there is only one edge between each vertex, and all of the vertices are connected. The tree is connected and acyclic.
- The cost of a spanning tree is the sum of the weights of the edges.
- Minimum spanning tree is the spanning tree with the smallest cost.
- Spanning tree with N vertices will have N-1 edges.
- Used in networks, laying wires for electricity/telephones, routing for internet connections, etc.




# Minimum Spanning Tree – Prim’s

# Prim’s Algorithm

- Prims algorithm analyzes all the connections between vertices and finds the set with minimum total weight that makes the graph connected.
- The vertices are divided into two sets:
  - S, the set of vertices in the spanning tree
  - And V-S, the remaining vertices
- Next, we choose the edge with the smallest weight that connects a vertex in S to a vertex in V-S and add it to the minimum spanning tree.

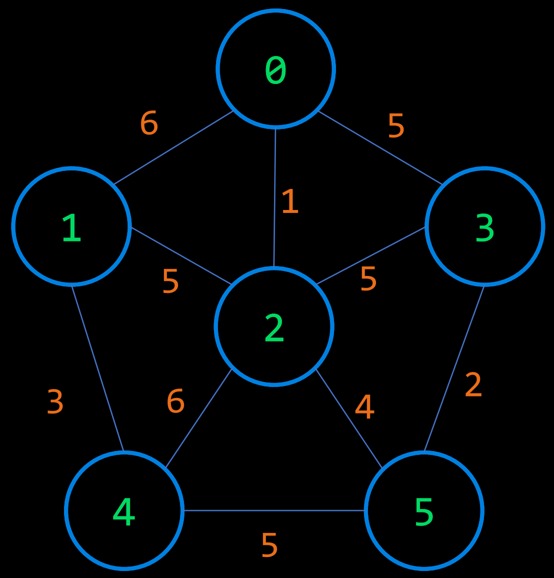


# Prim’s Algorithm

V

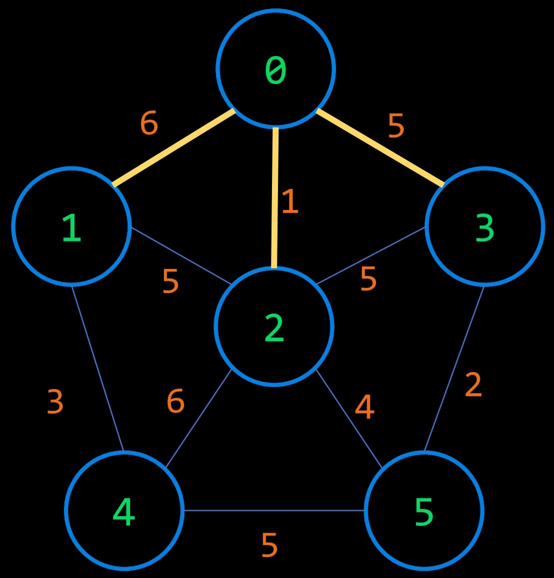
V-S

0 1 2 3 4 5



# Prim's Algorithm

| V | V-S |
|---|---|
| 0 | 1 2 3 4 5 |



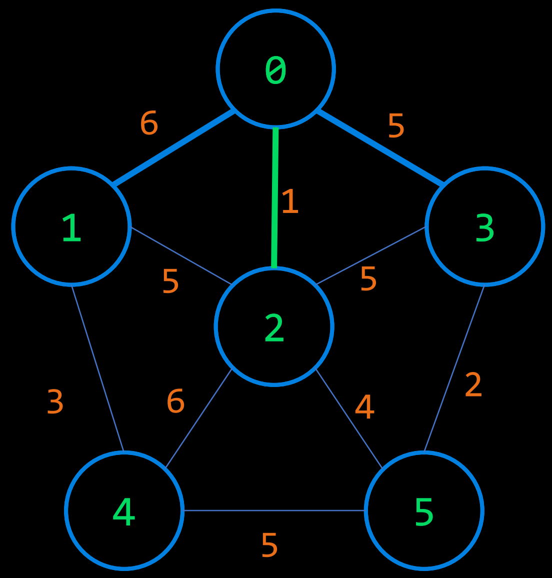
# Prim’s Algorithm

V

0 2

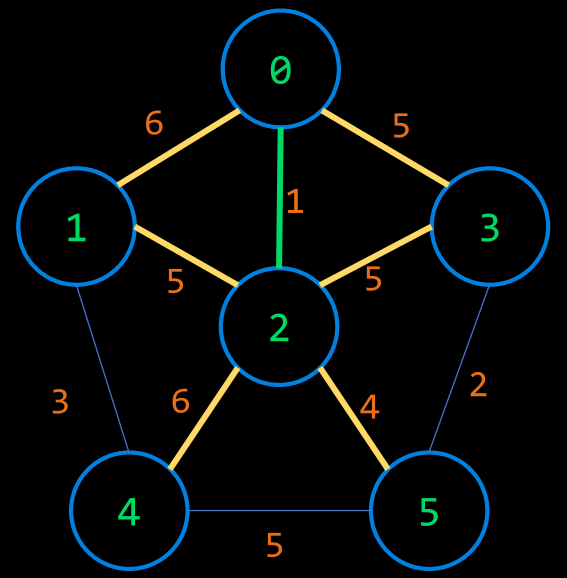
V-S

1 3 4 5



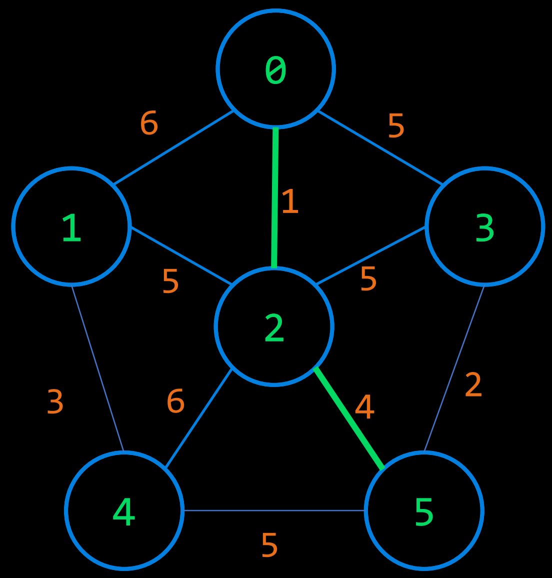
# Prim's Algorithm

| V | V-S |
|---|---|
| 0 2 | 1 3 4 5 |



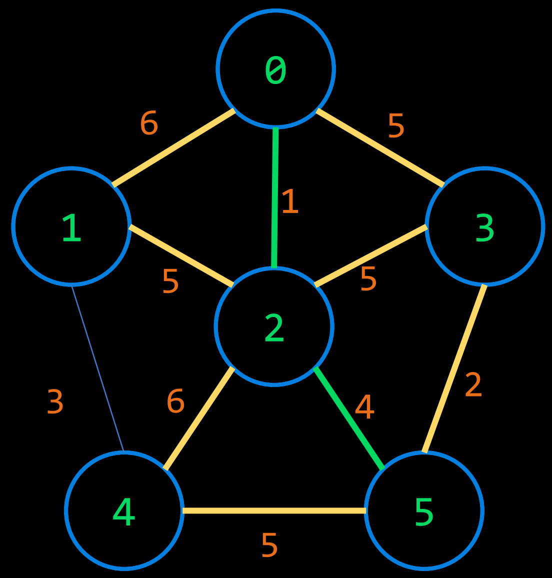
# Prim’s Algorithm

| V | V-S |
|---|---|
| 0 2 5 | 1 3 4 |



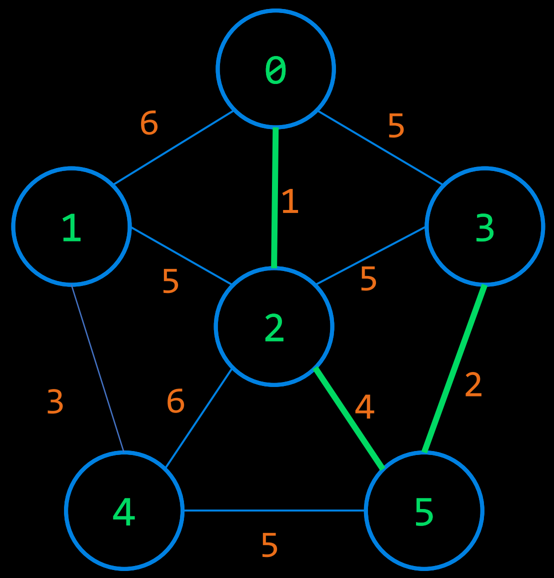
# Prim’s Algorithm

| V | V-S |
|---|---|
| 0 2 5 | 1 3 4 |



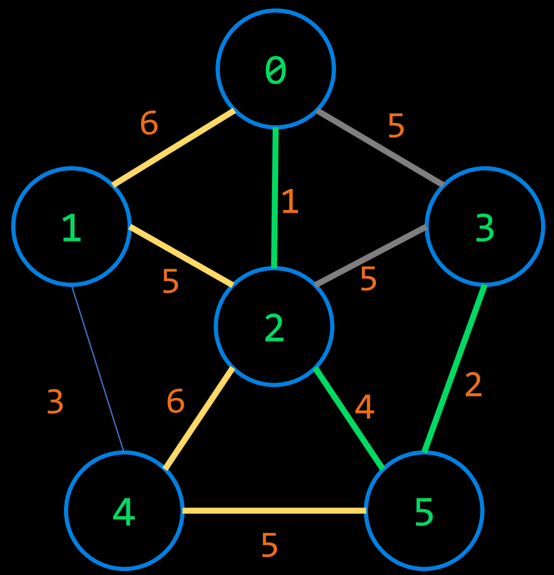
# Prim’s Algorithm

| V | V-S |
|---|---|
| 0 2 3 5 | 1 4 |



# Prim’s Algorithm

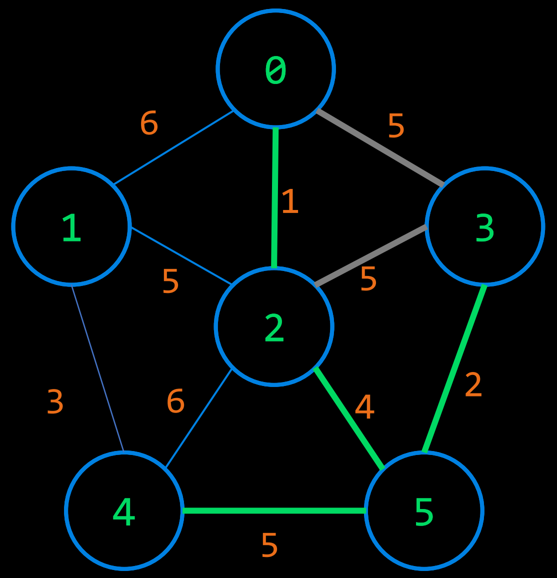
| V | V-S |
|---|---|
| 0 2 3 5 | 1 4 |




We choose the edge with the smallest weight that connects a vertex in S to a vertex in V-S and add it to the minimum spanning tree. Option to pick 1- or 4-5. Pick any.

# Prim's Algorithm

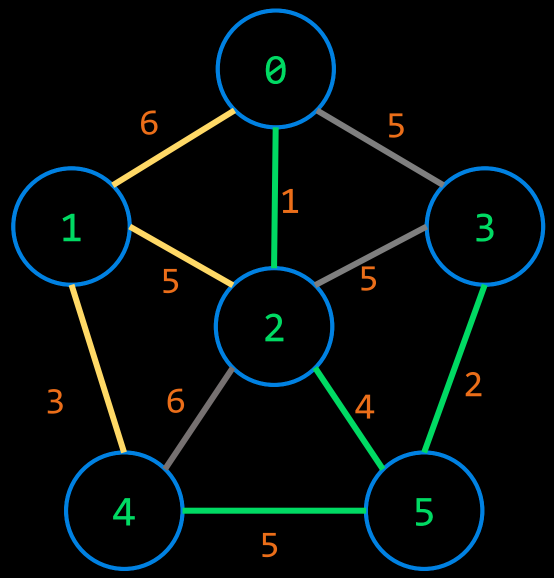
| V | V-S |
|---|---|
| 0 2 3 4 5 | 1 |



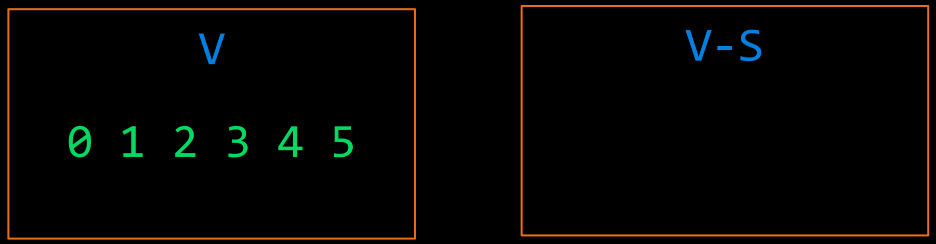

Pick 4-5.

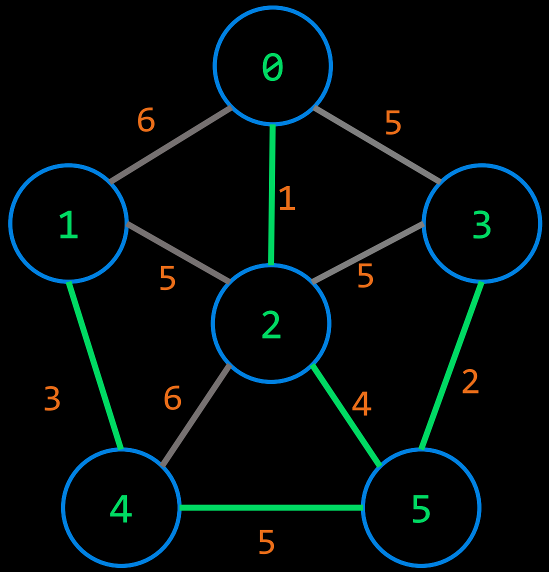
# Prim's Algorithm

| V | V-S |
|---|---|
| 0 2 3 4 5 | 1 |



# Prim’s Algorithm

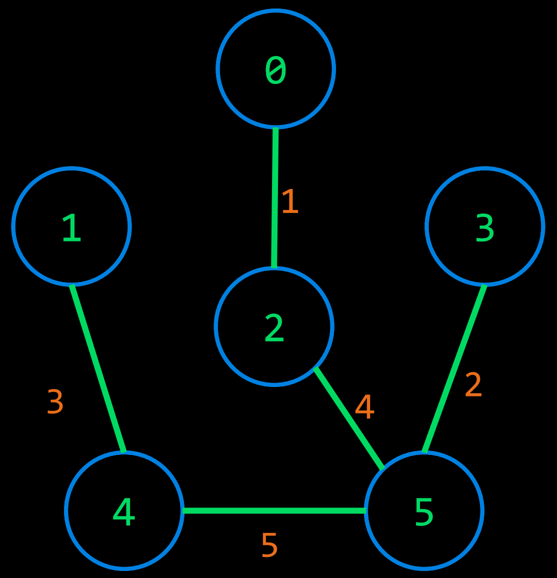



Pick 1-4.

# Prim’s Algorithm

| V | V-S |
|---|---|
| 0 1 2 3 4 5 | |




Sum of MST = 15.

# Prim’s Algorithm

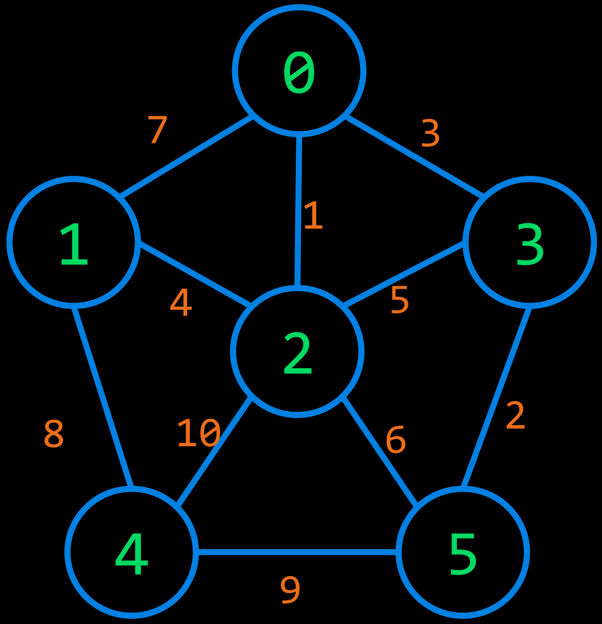
```
Input: An undirected, connected, weighted graph G.
Output: T, a minimum spanning tree for G.

T := Ø.
Pick any vertex in G and add it to T.

For j = 1 to n-1
      Let C be the set of edges with one endpoint inside T and
one endpoint outside T.
      Let e be a minimum weight edge in C.
      Add e to T.
      Add the endpoint of e not already in T to T.
End-for

Complexity: O(EV) or O(E log V) – using priority queues
```

# Minimum Spanning Tree – Kruskal’s

# Kruskal's Algorithm

**Arrange edges in ascending order**

| | |
|---|---|
| 0-2 | 1 |
| 3-5 | 2 |
| 0-3 | 3 |
| 1-2 | 4 |
| 2-3 | 5 |
| 2-5 | 6 |
| 0-1 | 7 |
| 1-4 | 8 |
| 4-5 | 9 |
| 2-4 | 10 |



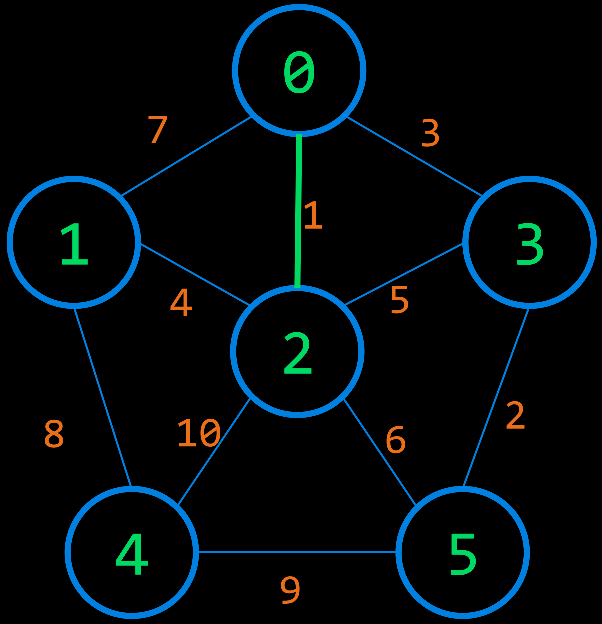

**Add edges in order as long as they don't create a cycle**

# Kruskal’s Algorithm

**Arrange edges in ascending order**

| | |
|---|---|
| 0-2 | 1 |
| 3-5 | 2 |
| 0-3 | 3 |
| 1-2 | 4 |
| 2-3 | 5 |
| 2-5 | 6 |
| 0-1 | 7 |
| 1-4 | 8 |
| 4-5 | 9 |
| 2-4 | 10 |

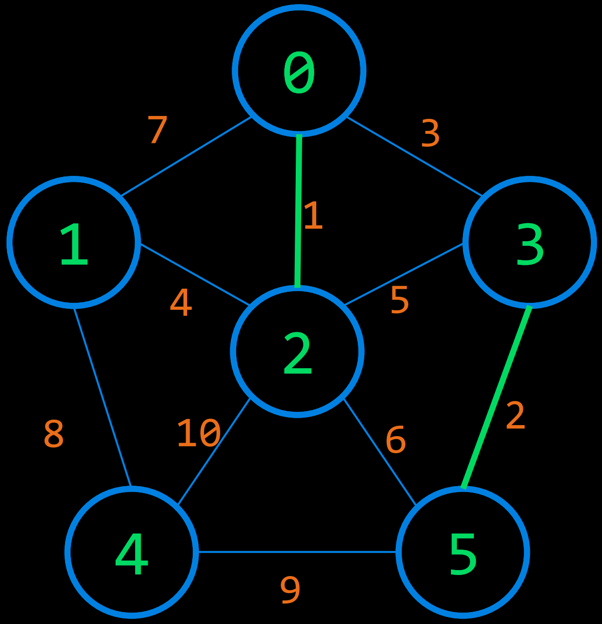
**Add edges in order as long as they don’t create a cycle**



# Kruskal's Algorithm

## Arrange edges in ascending order

| | |
|---|---|
| 0-2 | 1 |
| 3-5 | 2 |
| 0-3 | 3 |
| 1-2 | 4 |
| 2-3 | 5 |
| 2-5 | 6 |
| 0-1 | 7 |
| 1-4 | 8 |
| 4-5 | 9 |
| 2-4 | 10 |

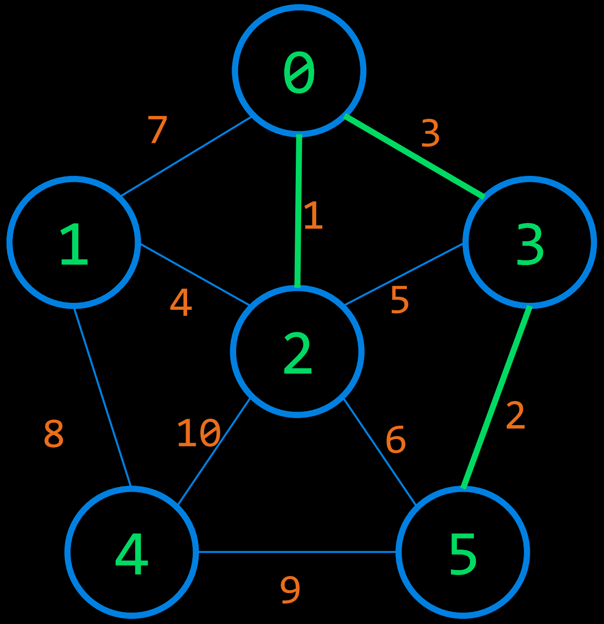



## Add edges in order as long as they don't create a cycle

# Kruskal’s Algorithm

**Arrange edges in ascending order**

| | |
|---|---|
| 0-2 | 1 |
| 3-5 | 2 |
| 0-3 | 3 |
| 1-2 | 4 |
| 2-3 | 5 |
| 2-5 | 6 |
| 0-1 | 7 |
| 1-4 | 8 |
| 4-5 | 9 |
| 2-4 | 10 |

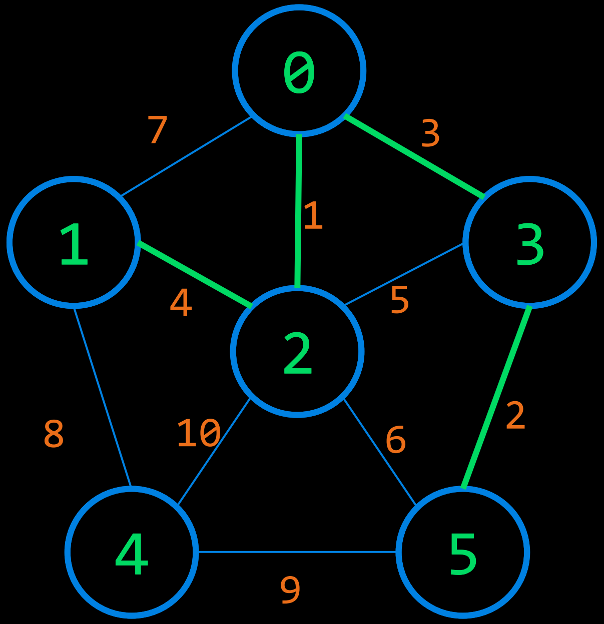



**Add edges in order as long as they don’t create a cycle**

# Kruskal's Algorithm

Arrange edges in ascending order

| | |
|---|---|
| 0-2 | 1 |
| 3-5 | 2 |
| 0-3 | 3 |
| 1-2 | 4 |
| 2-3 | 5 |
| 2-5 | 6 |
| 0-1 | 7 |
| 1-4 | 8 |
| 4-5 | 9 |
| 2-4 | 10 |

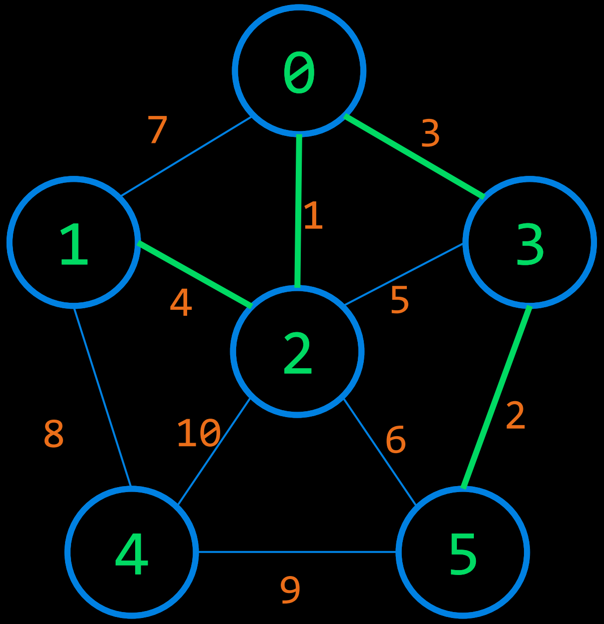



Add edges in order as long as they don't create a cycle

# Kruskal’s Algorithm

Arrange edges in ascending order

| | |
|---|---|
| 0-2 | 1 |
| 3-5 | 2 |
| 0-3 | 3 |
| 1-2 | 4 |
| ~~2-3~~ | ~~5~~ |
| 2-5 | 6 |
| 0-1 | 7 |
| 1-4 | 8 |
| 4-5 | 9 |
| 2-4 | 10 |

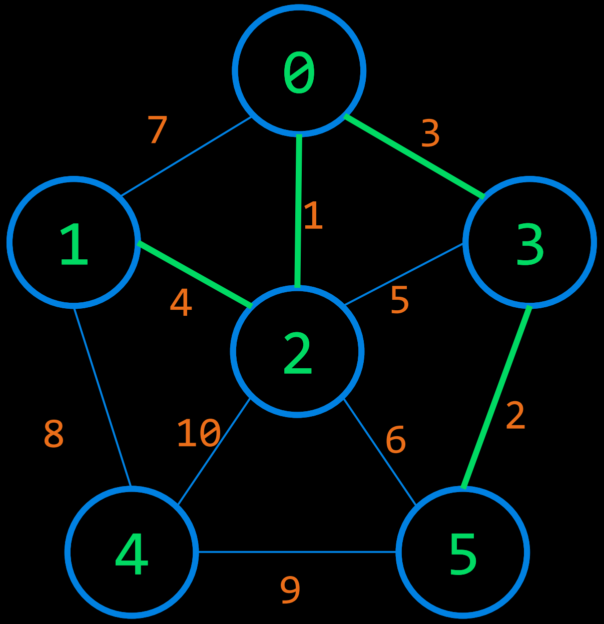



Add edges in order as long as they don’t create a cycle

# Kruskal's Algorithm

Arrange edges in ascending order

| | |
|---|---|
| 0-2 | 1 |
| 3-5 | 2 |
| 0-3 | 3 |
| 1-2 | 4 |
| ~~2-3~~ | ~~5~~ |
| ~~2-5~~ | ~~6~~ |
| 0-1 | 7 |
| 1-4 | 8 |
| 4-5 | 9 |
| 2-4 | 10 |

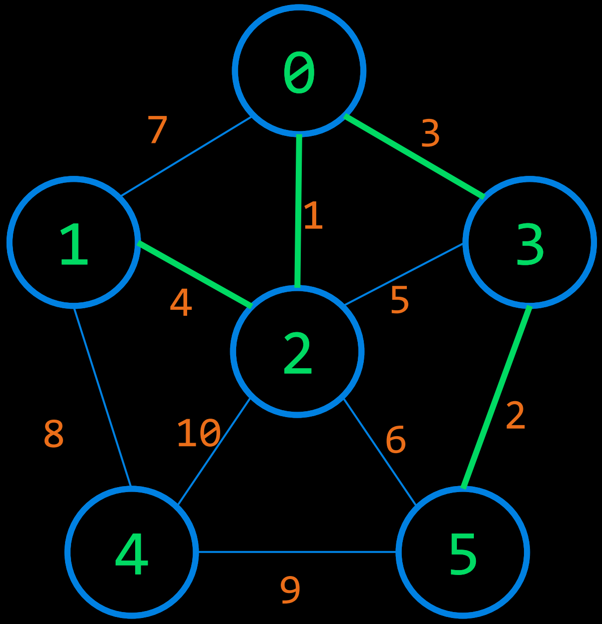



Add edges in order as long as they don't create a cycle

# Kruskal’s Algorithm

Arrange edges in ascending order

| | |
|---|---|
| 0-2 | 1 |
| 3-5 | 2 |
| 0-3 | 3 |
| 1-2 | 4 |
| ~~2-3~~ | ~~5~~ |
| ~~2-5~~ | ~~6~~ |
| ~~0-1~~ | ~~7~~ |
| 1-4 | 8 |
| 4-5 | 9 |
| 2-4 | 10 |

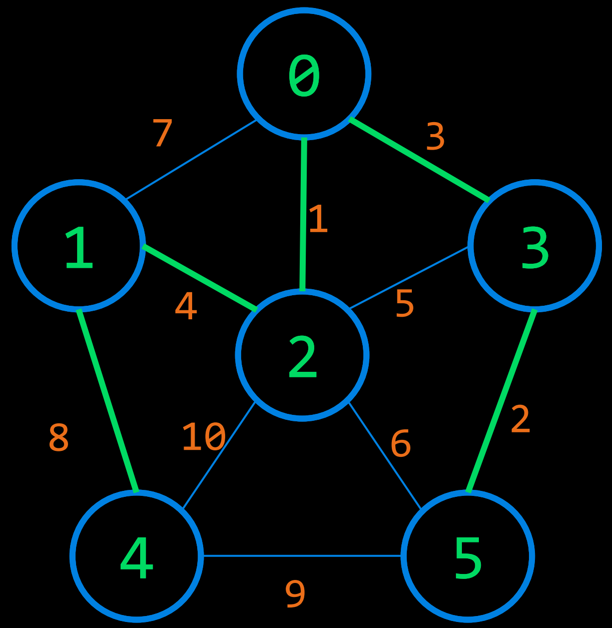



Add edges in order as long as they don’t create a cycle

# Kruskal’s Algorithm

Arrange edges in ascending order

| | |
|---|---|
| 0-2 | 1 |
| 3-5 | 2 |
| 0-3 | 3 |
| 1-2 | 4 |
| ~~2-3~~ | ~~5~~ |
| ~~2-5~~ | ~~6~~ |
| ~~0-1~~ | ~~7~~ |
| 1-4 | 8 |
| 4-5 | 9 |
| 2-4 | 10 |

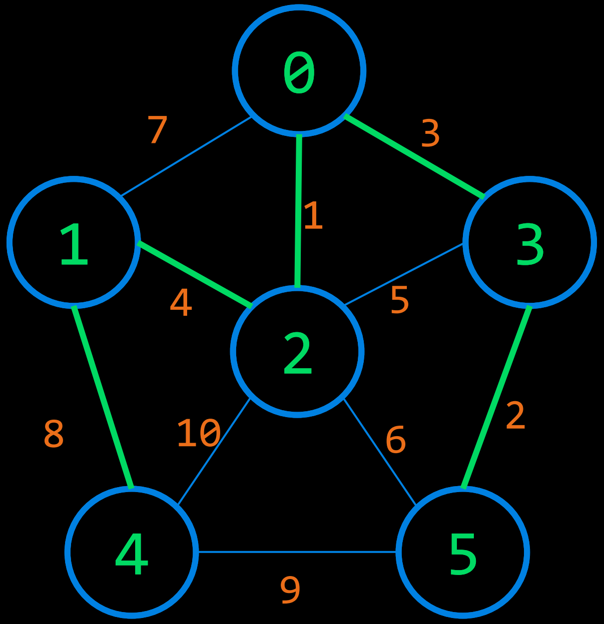



Add edges in order as long as they don’t create a cycle

# Kruskal's Algorithm

## Arrange edges in ascending order

| | |
|---|---|
| 0-2 | 1 |
| 3-5 | 2 |
| 0-3 | 3 |
| 1-2 | 4 |
| ~~2-3~~ | ~~5~~ |
| ~~2-5~~ | ~~6~~ |
| ~~0-1~~ | ~~7~~ |
| 1-4 | 8 |
| ~~4-5~~ | ~~9~~ |
| 2-4 | 10 |

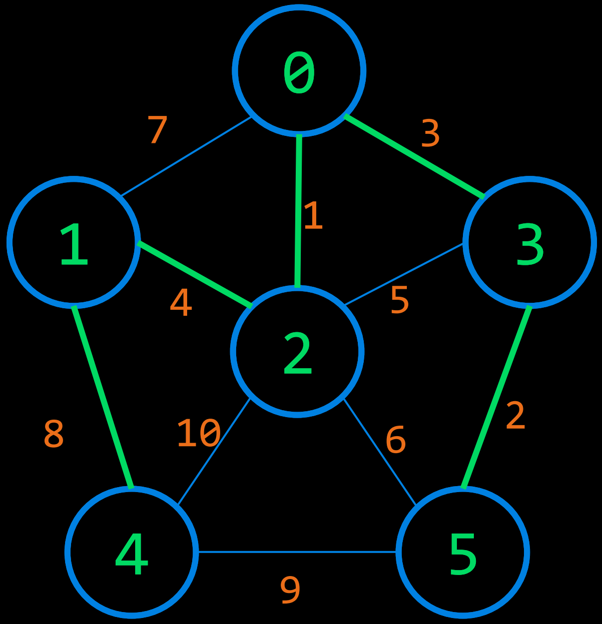



## Add edges in order as long as they don't create a cycle

# Kruskal's Algorithm

## Arrange edges in ascending order

| | |
|---|---|
| 0-2 | 1 |
| 3-5 | 2 |
| 0-3 | 3 |
| 1-2 | 4 |
| ~~2-3~~ | ~~5~~ |
| ~~2-5~~ | ~~6~~ |
| ~~0-1~~ | ~~7~~ |
| 1-4 | 8 |
| ~~4-5~~ | ~~9~~ |
| ~~2-4~~ | ~~10~~ |

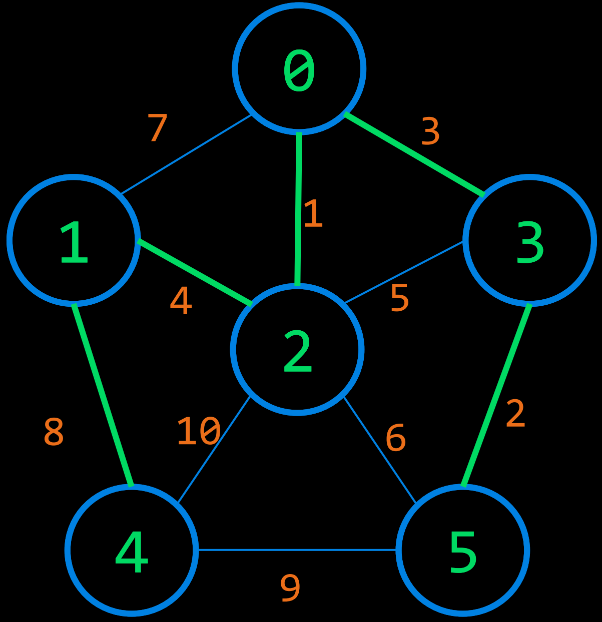



## Add edges in order as long as they don't create a cycle

# Kruskal’s Algorithm

## Arrange edges in ascending order

0-2 1
3-5 2
0-3 3
1-2 4
~~2-3 5~~
~~2-5 6~~
~~0-1 7~~
1-4 8
~~4-5 9~~
~~2-4 10~~




**Minimum Spanning Tree Sum = 18**

# Questions

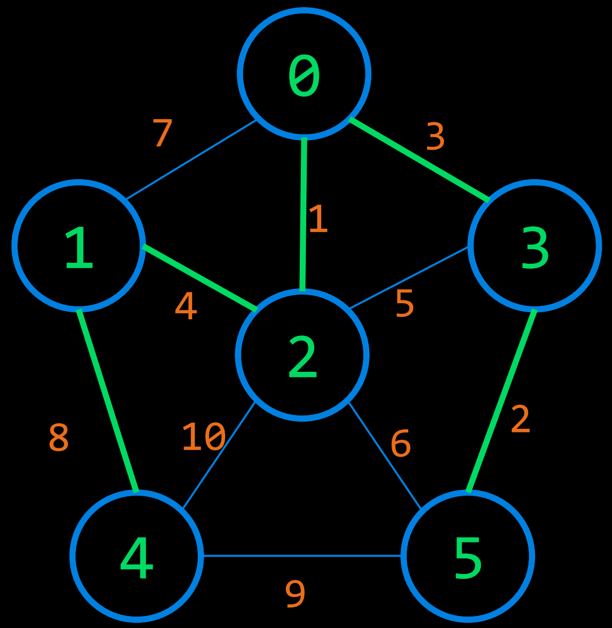
# Kruskal's Algorithm

**How can we detect a cycle when adding an edge?**

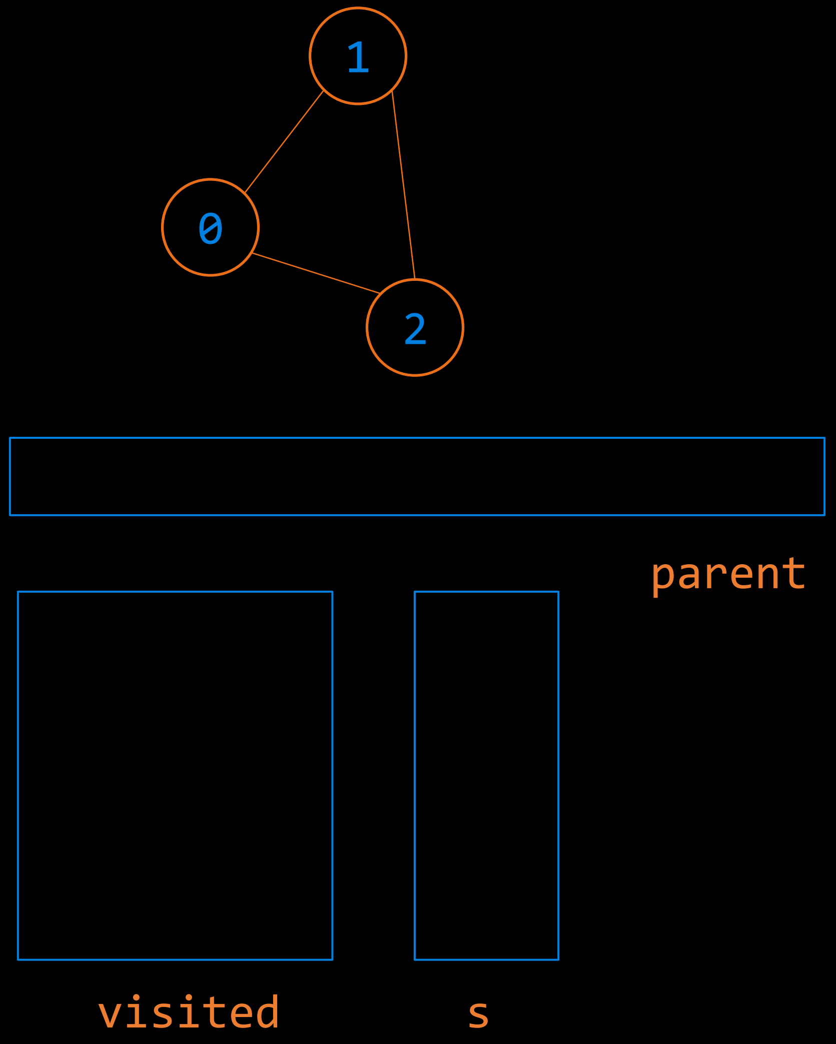
Method 1:

Cycle Detection using DFS. Find back edges.

Back Edge: An edge that connects an ancestor during DFS traversal.

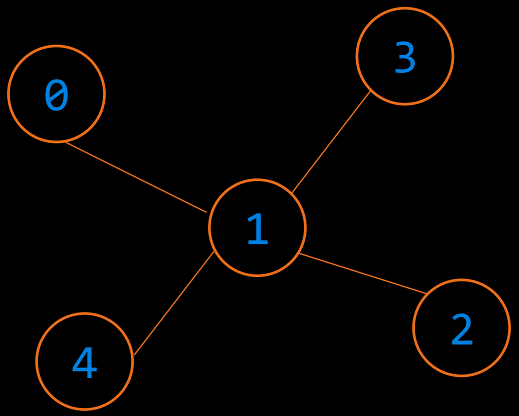


# 7.3.1 Detect whether there is a Cycle in an Undirected Graph




```
bool anyCycle(const Graph& graph)
{
    set<int> visited;
    vector<int> parent(graph.numVertices, -1);
    stack<int> s;
    visited.insert(0);
    s.push(0);
    while(!s.empty())
    {
            int u = s.top();
            s.pop();
            for(auto v: graph.adjList[u])
            {
                if ((visited.find(v)==visited.end()))
                {
                    visited.insert(v);
                    s.push(v);
                    parent[v] = u;
                }
                else if (parent[u] != v)
                    return true;
            }
     }
     return false;
}
```

# 7.3.1 Detect whether there is a Cycle in an Undirected Graph



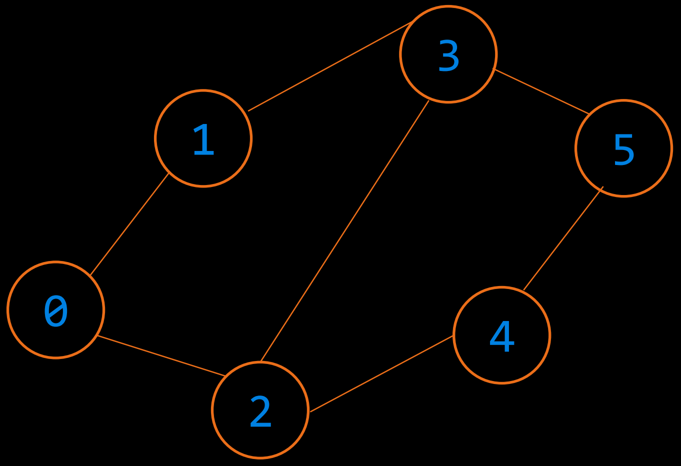

parent

visited

s

```
bool anyCycle(const Graph& graph)
{
    set<int> visited;
    vector<int> parent(graph.numVertices, -1);
    stack<int> s;
    visited.insert(0);
    s.push(0);
    while(!s.empty())
    {
            int u = s.top();
            s.pop();
            for(auto v: graph.adjList[u])
            {
                if ((visited.find(v)==visited.end()))
                {
                    visited.insert(v);
                    s.push(v);
                    parent[v] = u;
                }
                else if (parent[u] != v)
                    return true;
            }
      }
      return false;
}
```

# 7.3.1 Detect whether there is a Cycle in an Undirected Graph



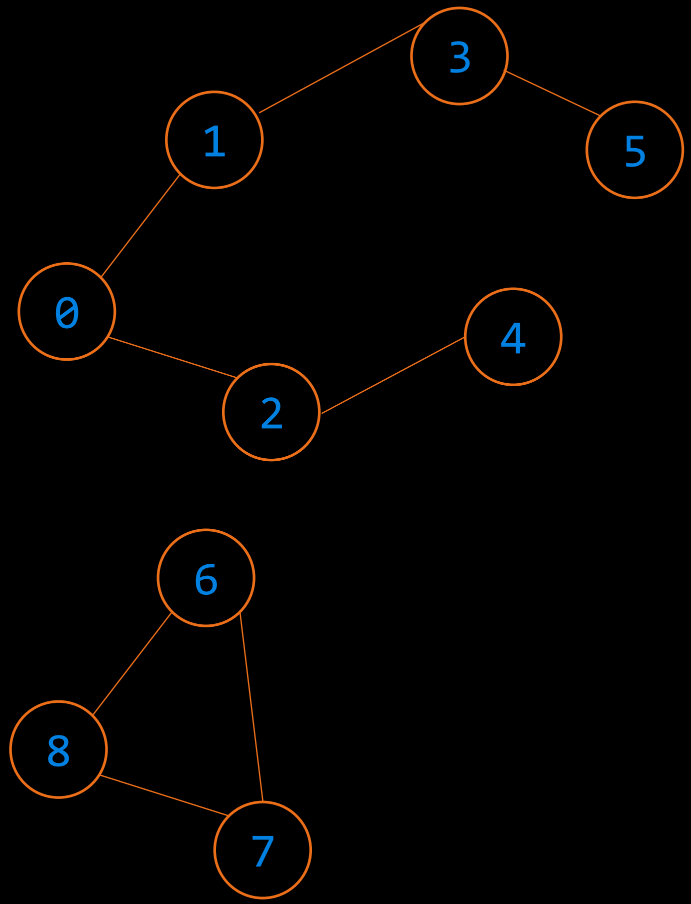

parent

visited

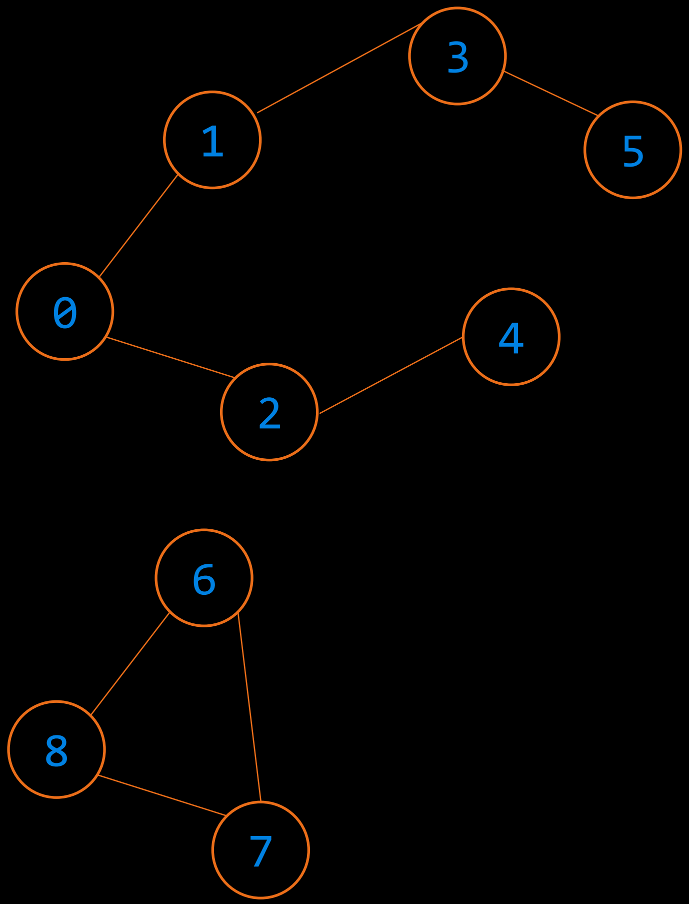
s

```
bool anyCycle(const Graph& graph)
{
    set<int> visited;
    vector<int> parent(graph.numVertices, -1);
    stack<int> s;
    visited.insert(0);
    s.push(0);
    while(!s.empty())
    {
            int u = s.top();
            s.pop();
            for(auto v: graph.adjList[u])
            {
                if ((visited.find(v)==visited.end()))
                {
                    visited.insert(v);
                    s.push(v);
                    parent[v] = u;
                }
                else if (parent[u] != v)
                    return true;
            }
    }
    return false;
}
```

# 7.3.1 Detect whether there is a Cycle in an Undirected Graph



# 7.3.1 Detect whether there is a Cycle in an Undirected Graph




```cpp
bool anyCycle(const Graph& graph)
{
    set<int> visited;
    vector<int> parent(graph.numVertices, -1);
    stack<int> s;
    for(int i=0; i<graph.numVertices; i++)
    {
        if ((visited.find(i)==visited.end()))
        {
            visited.insert(i);
            s.push(i);
            while(!s.empty())
            {
                int u = s.top();
                s.pop();
                for(auto v: graph.adjList[u])
                {
                    if ((visited.find(v)==visited.end()))
                    {
                        visited.insert(v);
                        s.push(v);
                        parent[v] = u;
                    }
                    else if (parent[u] != v)
                        return true;
                }
            }
        }
    }
    return false;
}
```

# Kruskal’s Algorithm

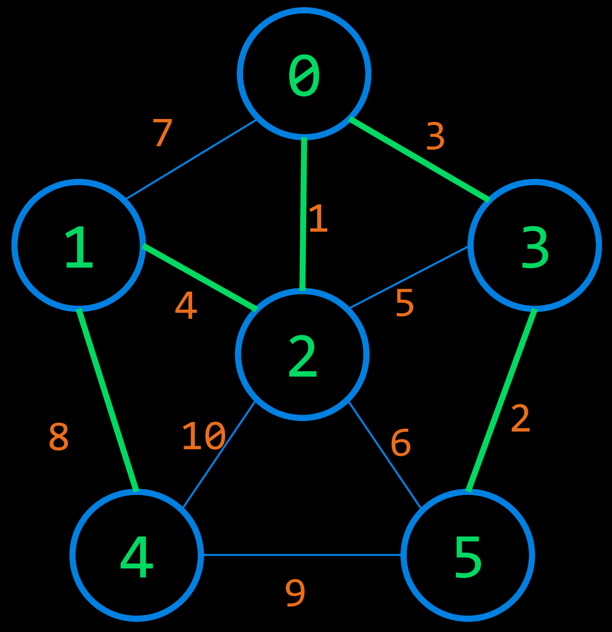
**How can we detect a cycle when adding an edge?**

Method 1:

Cycle Detection using DFS.

Works correctly but is computationally more expensive.

Complexity: O(E (V+E))

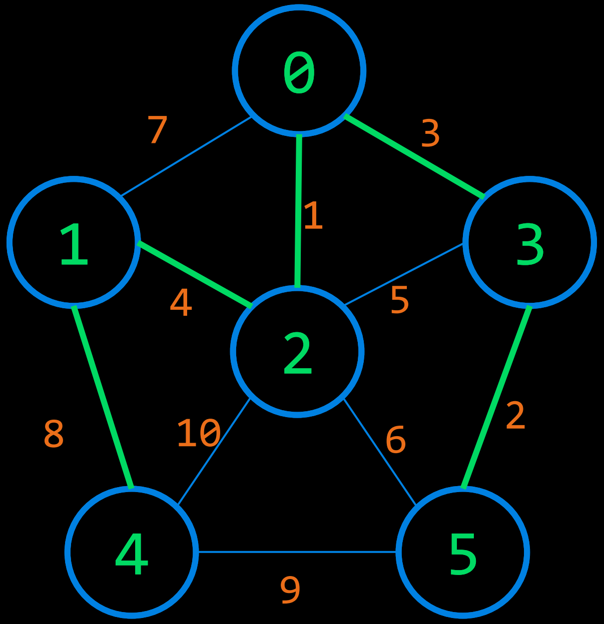


# Kruskal’s Algorithm

## How can we detect a cycle when adding an edge?

Method 2a:

- Create an empty set, S.
- For each edge, E:
    - If either of the vertices connecting E is not a part of the set, add the vertices of E to S
    - If, both the vertices are part of the set S, **ignore the edge as it forms a cycle.**

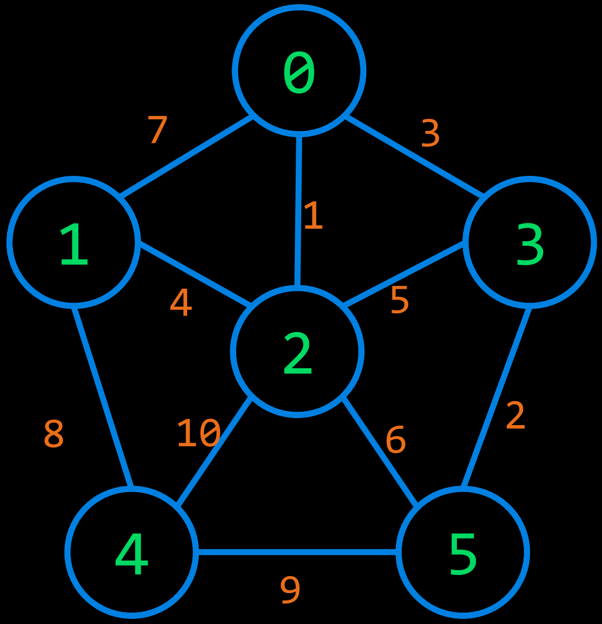


# Kruskal's Algorithm

**Arrange edges in ascending order**

| | |
|---|---|
| 0-2 | 1 |
| 3-5 | 2 |
| 0-3 | 3 |
| 1-2 | 4 |
| 2-3 | 5 |
| 2-5 | 6 |
| 0-1 | 7 |
| 1-4 | 8 |
| 4-5 | 9 |
| 2-4 | 10 |

**Add edges in order as long as they don't create a cycle**



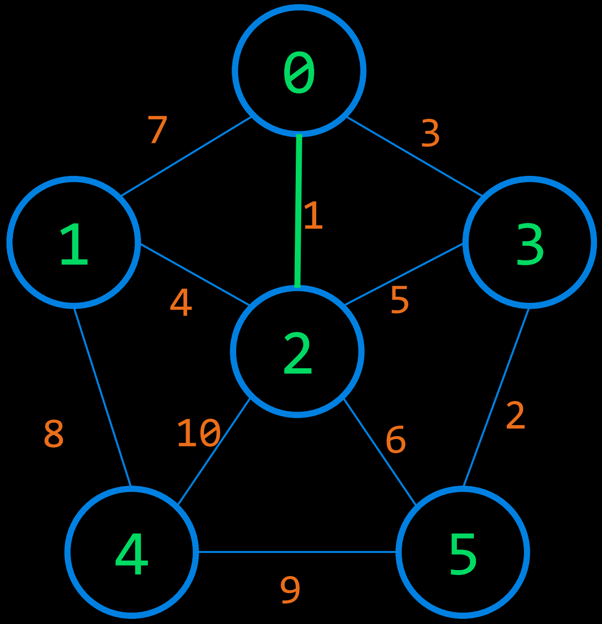
# Kruskal's Algorithm

## Arrange edges in ascending order

| | |
|---|---|
| 0-2 | 1 |
| 3-5 | 2 |
| 0-3 | 3 |
| 1-2 | 4 |
| 2-3 | 5 |
| 2-5 | 6 |
| 0-1 | 7 |
| 1-4 | 8 |
| 4-5 | 9 |
| 2-4 | 10 |

Set

0 2




## Add edges in order as long as they don't create a cycle

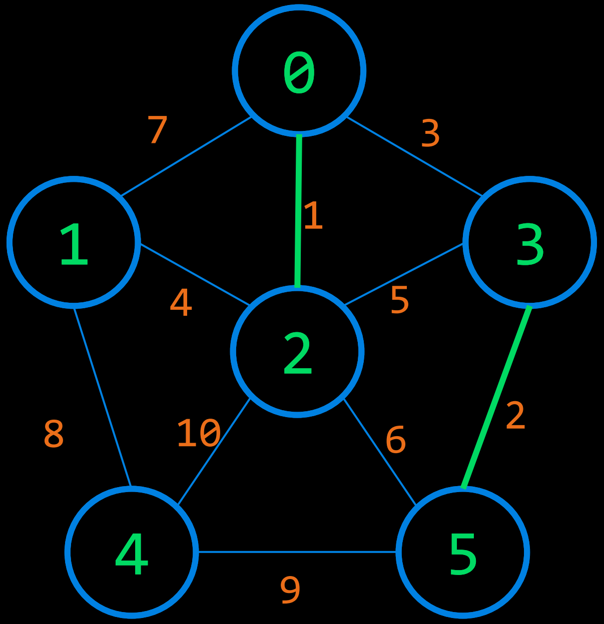
# Kruskal’s Algorithm

Arrange edges in ascending order

| | |
|---|---|
| 0-2 | 1 |
| 3-5 | 2 |
| 0-3 | 3 |
| 1-2 | 4 |
| 2-3 | 5 |
| 2-5 | 6 |
| 0-1 | 7 |
| 1-4 | 8 |
| 4-5 | 9 |
| 2-4 | 10 |

Set

0 2 3 5




Add edges in order as long as they don’t create a cycle

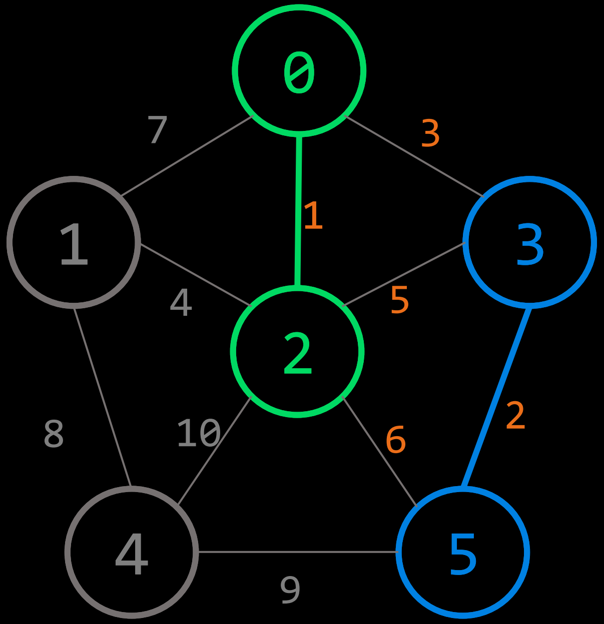
# Kruskal’s Algorithm

Arrange edges in ascending order

| | |
|---|---|
| 0-2 | 1 |
| 3-5 | 2 |
| 0-3 | 3 |
| 1-2 | 4 |
| 2-3 | 5 |
| 2-5 | 6 |
| 0-1 | 7 |
| 1-4 | 8 |
| 4-5 | 9 |
| 2-4 | 10 |

Set

0 2 3 5




Disconnected Components

Add edges in order as long as they don’t create a cycle

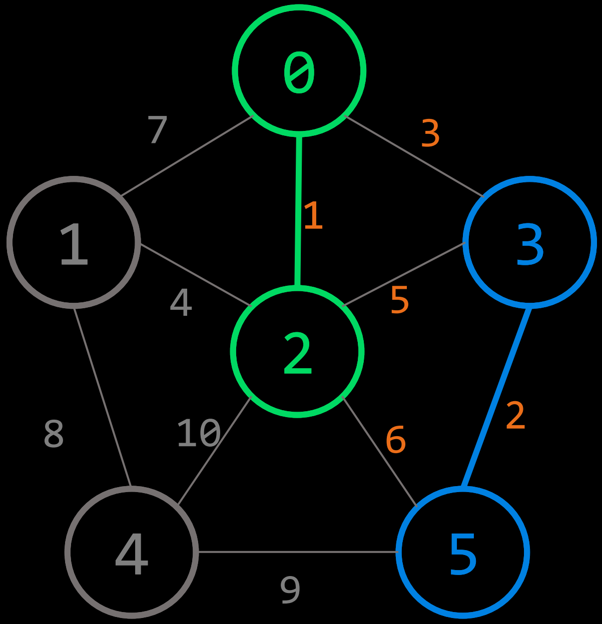
# Kruskal's Algorithm

Arrange edges in ascending order

| | |
|---|---|
| 0-2 | 1 |
| 3-5 | 2 |
| 0-3 | 3 |
| 1-2 | 4 |
| 2-3 | 5 |
| 2-5 | 6 |
| 0-1 | 7 |
| 1-4 | 8 |
| 4-5 | 9 |
| 2-4 | 10 |

Set

0 2 3 5




Disconnected Components
0-3, 2-3, 2-5 will never be picked in MST

Add edges in order as long as they don't create a cycle

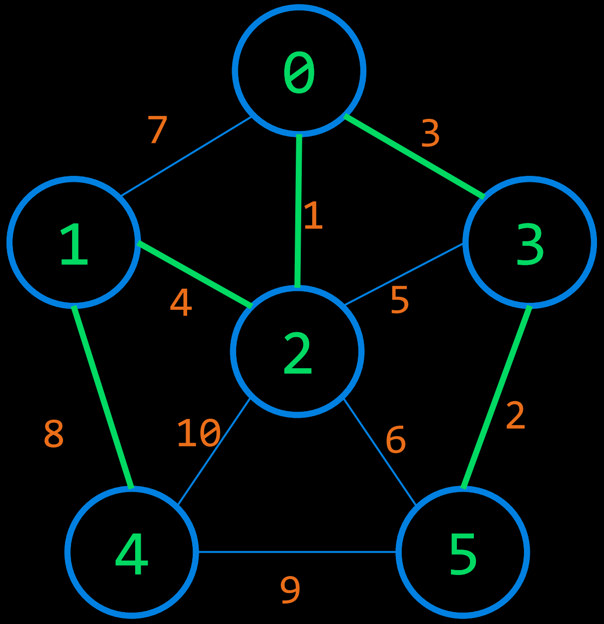
# Kruskal's Algorithm

**How can we detect a cycle when adding an edge?**

Method 2a:

- Create an empty set, S.
- For each edge, E:
  - If either of the vertices connecting E is not a part of the set, add the vertices of E to S
  - If, both the vertices are part of the set S, ignore the edge as it forms a cycle.

This will not work whenever we pick edges in an order such that we have two disconnected components. Adding edges leads to connected components!



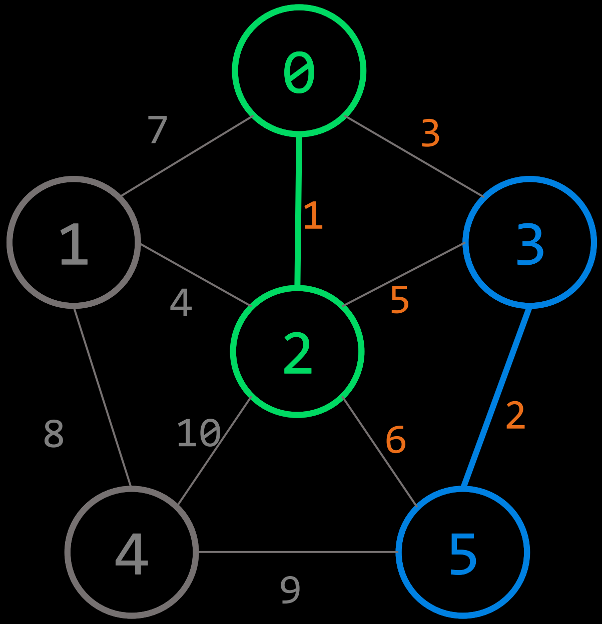
# Kruskal's Algorithm

## How can we detect a cycle when adding an edge?

Method 2b:

Disjoint Sets – Weighted Union

- A group of sets. There is no item in common in any of the sets.
- Operations:
  - find(i) identify the set that contains i
  - union(i, j) merge the set that contains i and the set that contains j
- Disjoint sets represent connected components.
- A cycle is created by adding an edge for which both vertices are in the same connected component.
- Complexity: O(E log V)




Disconnected Components
Two connected components:
{0,2} and {3,5}

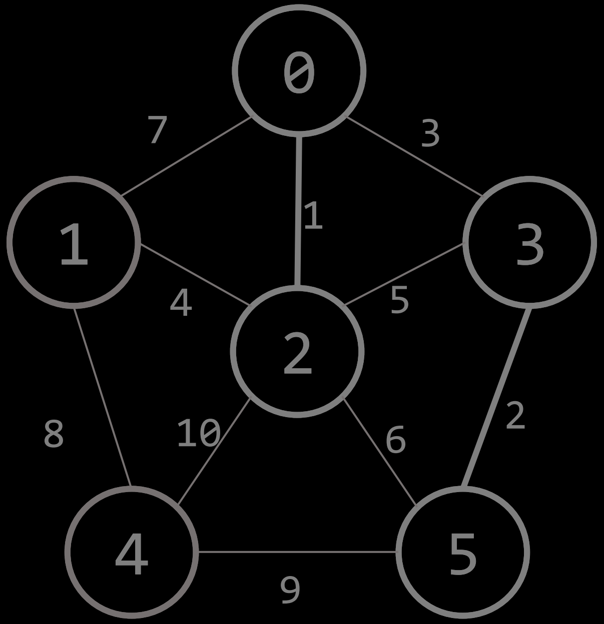
# Disjoint Sets

Disjoint Sets – Union/Find

- Optimally represented as an array where each index stores the parent of the “index” vertex. An entire set is represented as a tree.

- Operations:
  - union(i, j) merge the set that contains i and the set that contains j

  - find(i) identify the set that contains i

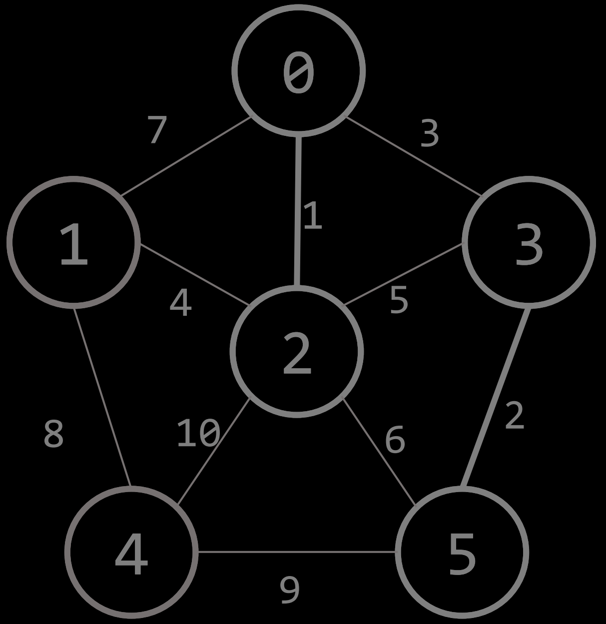


# Disjoint Sets

Disjoint Sets – Union/Find

- Optimally represented as an array where each index stores the parent of the “index” vertex. An entire set is represented as a tree.

- Operations:
  - union(i, j) merge the set that contains i and the set that contains j
    ```
    pi = find(i)
    pj = find(j)
    arr [pi] = pj
    ```
  - find(i) identify the set that contains i
    ```
    if(arr[i]) == -1)
            return i
    else
            return find(arr[i])
    ```



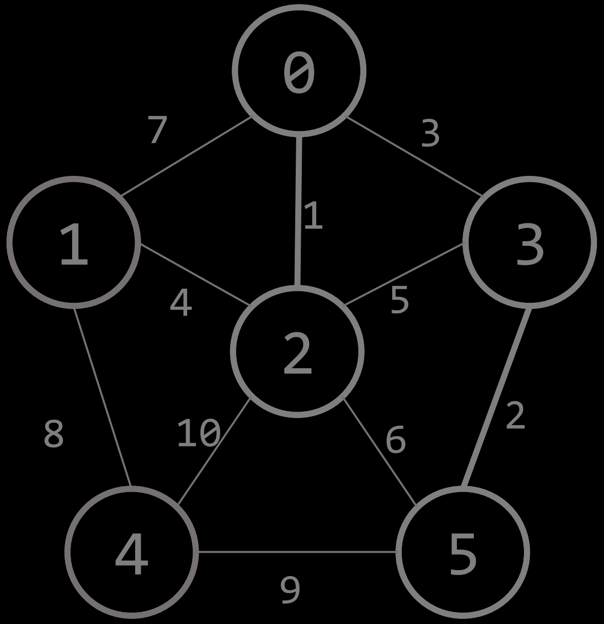
# Disjoint Sets

- Operations:
  - union(i, j) merge the set that contains i and the set that contains j

```
pi = find(i)
pj = find(j)
arr [pi] = pj
```

  - find(i) identify the set that contains i

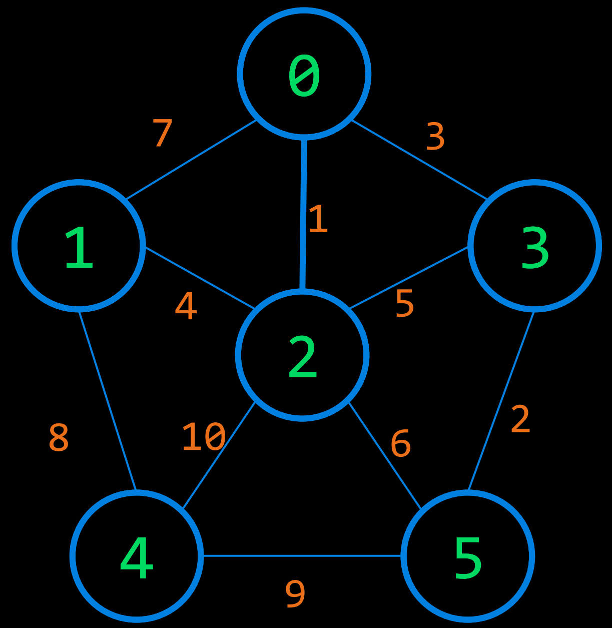
```
if(arr[i]) == -1)
        return i
else
        return find(arr[i])
```



# Kruskal's Algorithm

**Arrange edges in ascending order**

| | |
|---|---|
| 0-2 | 1 |
| 3-5 | 2 |
| 0-3 | 3 |
| 1-2 | 4 |
| 2-3 | 5 |
| 2-5 | 6 |
| 0-1 | 7 |
| 1-4 | 8 |
| 4-5 | 9 |
| 2-4 | 10 |



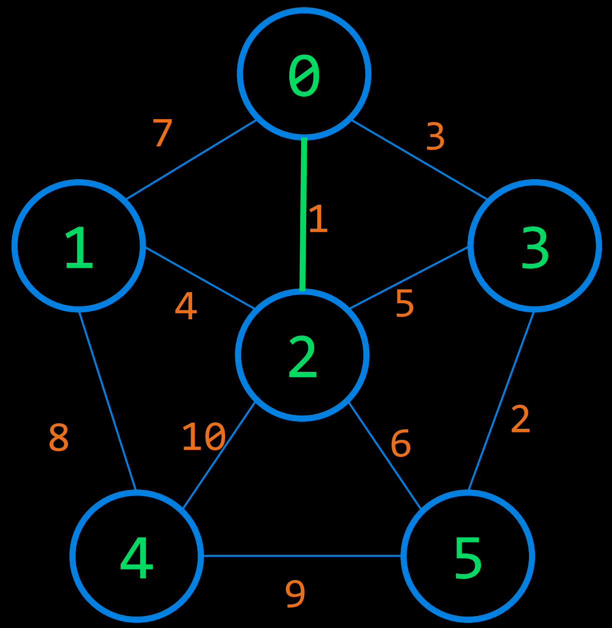

```
Kruskal's algorithm – choose an edge i-j
                      if find(i) != find(j)     //not in same disjoint set
                        union(i, j)   //add the edge, which joins the components
```

Disjoint sets: {0}, {1}, {2}, {3}, {4}, {5}

# Kruskal’s Algorithm

**Arrange edges in ascending order**

| | |
|---|---|
| 0-2 | 1 |
| 3-5 | 2 |
| 0-3 | 3 |
| 1-2 | 4 |
| 2-3 | 5 |
| 2-5 | 6 |
| 0-1 | 7 |
| 1-4 | 8 |
| 4-5 | 9 |
| 2-4 | 10 |



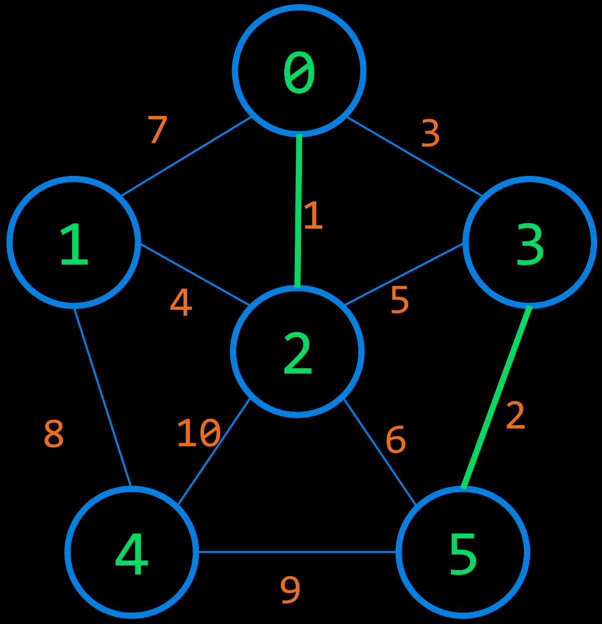

```
Kruskal’s algorithm – choose an edge i-j
                      if find(i) != find(j)    //not in same disjoint set
                         union(i, j)   //add the edge, which joins the components
```

Disjoint sets: {0, 2}, {1}, {3}, {4}, {5}

# Kruskal’s Algorithm

**Arrange edges in ascending order**

| | |
|---|---|
| 0-2 | 1 |
| 3-5 | 2 |
| 0-3 | 3 |
| 1-2 | 4 |
| 2-3 | 5 |
| 2-5 | 6 |
| 0-1 | 7 |
| 1-4 | 8 |
| 4-5 | 9 |
| 2-4 | 10 |



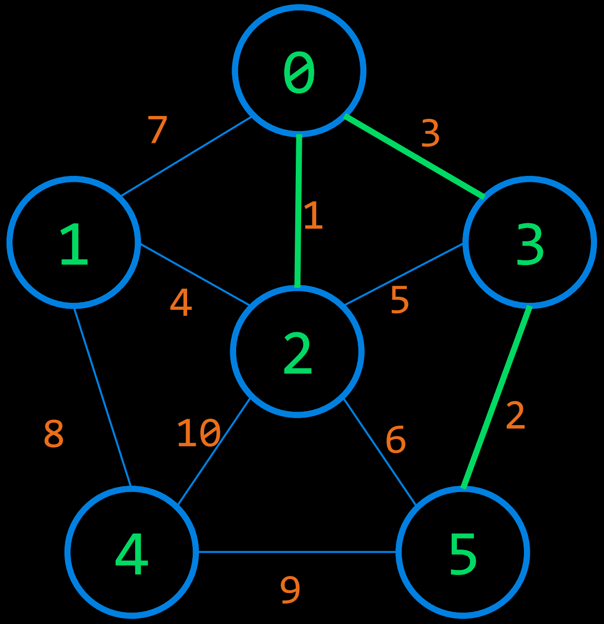

```
Kruskal’s algorithm – choose an edge i-j
                      if find(i) != find(j)    //not in same disjoint set
                        union(i, j)   //add the edge, which joins the components
```

Disjoint sets: {0, 2}, {1}, {3, 5}, {4}

# Kruskal's Algorithm

**Arrange edges in ascending order**

| | |
|---|---|
| 0-2 | 1 |
| 3-5 | 2 |
| 0-3 | 3 |
| 1-2 | 4 |
| 2-3 | 5 |
| 2-5 | 6 |
| 0-1 | 7 |
| 1-4 | 8 |
| 4-5 | 9 |
| 2-4 | 10 |



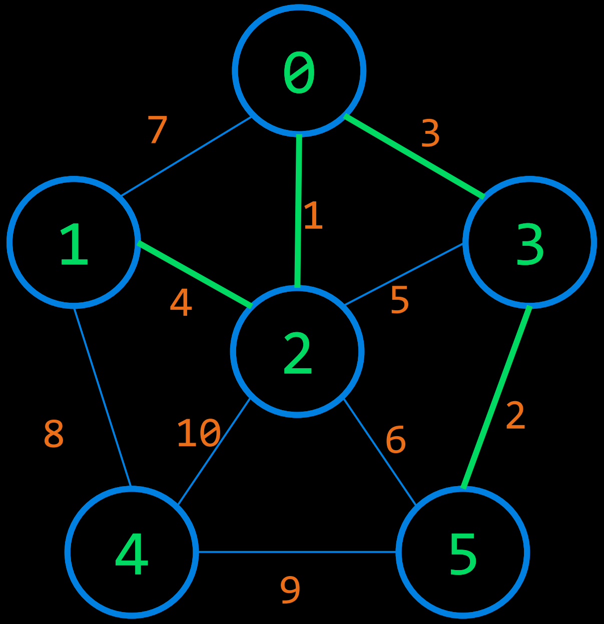

```
Kruskal's algorithm – choose an edge i-j
                      if find(i) != find(j)     //not in same disjoint set
                        union(i, j)    //add the edge, which joins the components
```

Disjoint sets: {0, 2, 3, 5}, {1}, {4}

# Kruskal's Algorithm

**Arrange edges in ascending order**

| | |
|---|---|
| 0-2 | 1 |
| 3-5 | 2 |
| 0-3 | 3 |
| 1-2 | 4 |
| 2-3 | 5 |
| 2-5 | 6 |
| 0-1 | 7 |
| 1-4 | 8 |
| 4-5 | 9 |
| 2-4 | 10 |



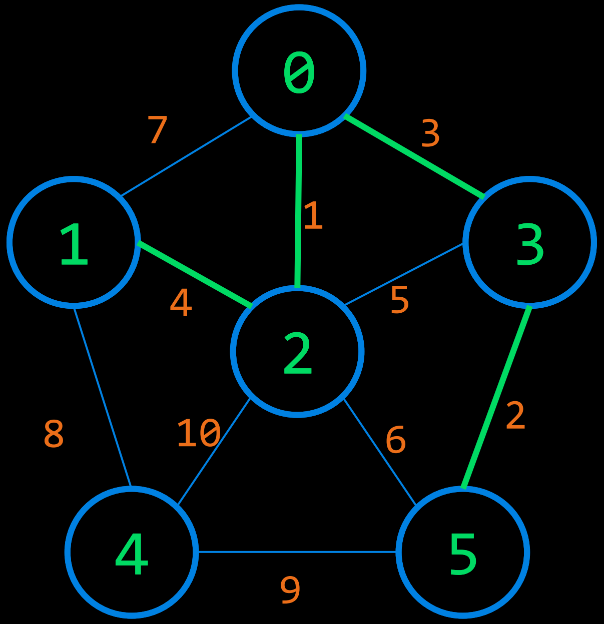

```
Kruskal's algorithm – choose an edge i-j
                      if find(i) != find(j)    //not in same disjoint set
                         union(i, j)   //add the edge, which joins the components
```

Disjoint sets: {0, 1, 2, 3, 5}, {4}

# Kruskal's Algorithm

**Arrange edges in ascending order**

| | |
|---|---|
| 0-2 | 1 |
| 3-5 | 2 |
| 0-3 | 3 |
| 1-2 | 4 |
| ~~2-3~~ | ~~5~~ |
| 2-5 | 6 |
| 0-1 | 7 |
| 1-4 | 8 |
| 4-5 | 9 |
| 2-4 | 10 |



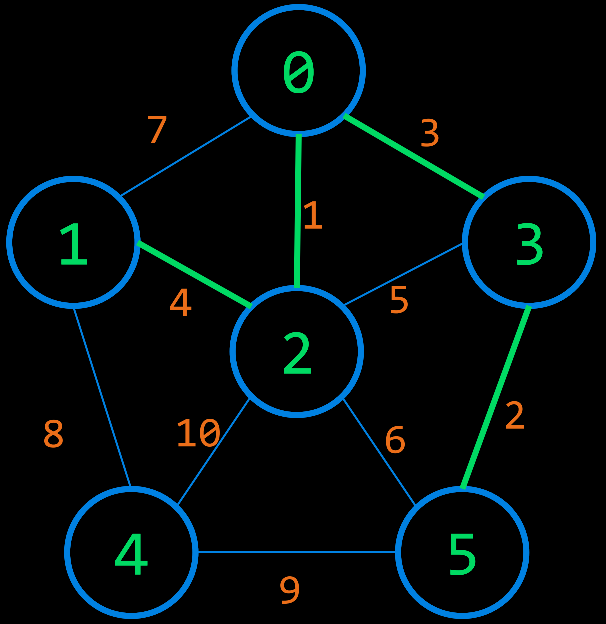

```
Kruskal's algorithm – choose an edge i-j
                     if find(i) != find(j)     //not in same disjoint set
                        union(i, j)    //add the edge, which joins the components
```

Disjoint sets: {0, 1, 2, 3, 5}, {4}

# Kruskal’s Algorithm

## Arrange edges in ascending order

| | |
|---|---|
| 0-2 | 1 |
| 3-5 | 2 |
| 0-3 | 3 |
| 1-2 | 4 |
| ~~2-3~~ | ~~5~~ |
| ~~2-5~~ | ~~6~~ |
| 0-1 | 7 |
| 1-4 | 8 |
| 4-5 | 9 |
| 2-4 | 10 |



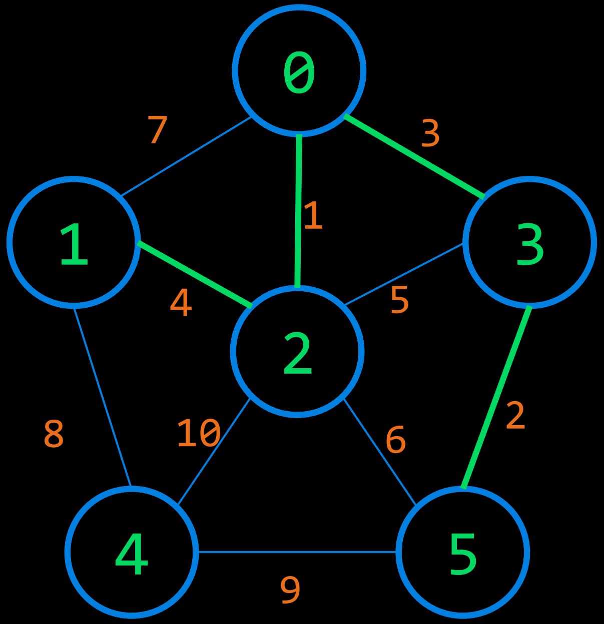

```
Kruskal’s algorithm – choose an edge i-j
                      if find(i) != find(j)     //not in same disjoint set
                         union(i, j)   //add the edge, which joins the components
```

Disjoint sets: {0, 1, 2, 3, 5}, {4}

# Kruskal’s Algorithm

**Arrange edges in ascending order**

```
0-2    1
3-5    2
0-3    3
1-2    4
```
~~2-3    5~~
~~2-5    6~~
~~0-1    7~~
```
1-4    8
4-5    9
2-4    10
```



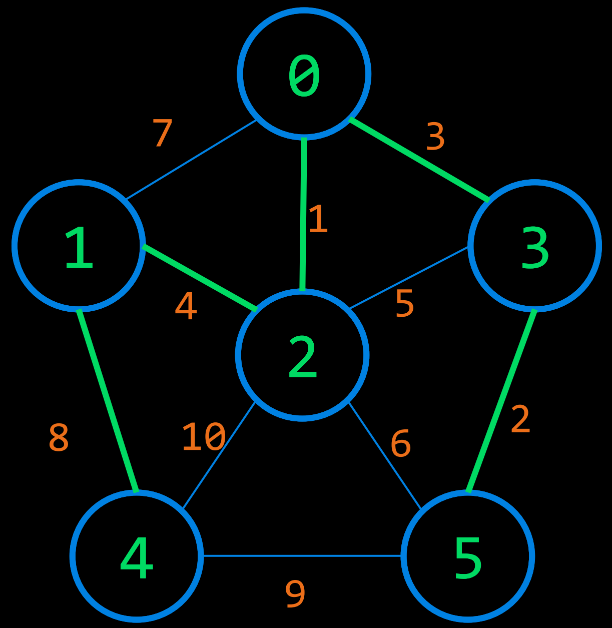

```
Kruskal’s algorithm – choose an edge i-j
                      if find(i) != find(j)    //not in same disjoint set
                         union(i, j)   //add the edge, which joins the components
```

Disjoint sets: {0, 1, 2, 3, 5}, {4}

# Kruskal's Algorithm

**Arrange edges in ascending order**

| | |
|---|---|
| 0-2 | 1 |
| 3-5 | 2 |
| 0-3 | 3 |
| 1-2 | 4 |
| ~~2-3~~ | ~~5~~ |
| ~~2-5~~ | ~~6~~ |
| ~~0-1~~ | ~~7~~ |
| 1-4 | 8 |
| 4-5 | 9 |
| 2-4 | 10 |



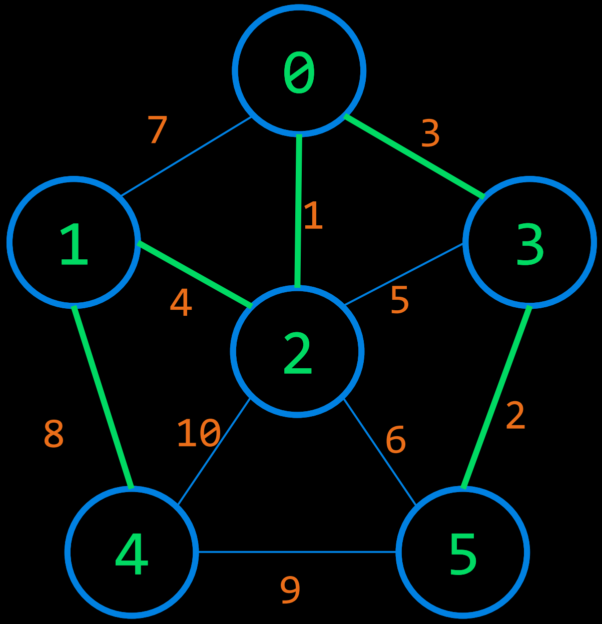

```
Kruskal's algorithm – choose an edge i-j
                      if find(i) != find(j)    //not in same disjoint set
                         union(i, j)   //add the edge, which joins the components
```

Disjoint sets: {0, 1, 2, 3, 4, 5}

# Kruskal's Algorithm

**Arrange edges in ascending order**

| | |
|---|---|
| 0-2 | 1 |
| 3-5 | 2 |
| 0-3 | 3 |
| 1-2 | 4 |
| ~~2-3~~ | ~~5~~ |
| ~~2-5~~ | ~~6~~ |
| ~~0-1~~ | ~~7~~ |
| 1-4 | 8 |
| ~~4-5~~ | ~~9~~ |
| ~~2-4~~ | ~~10~~ |



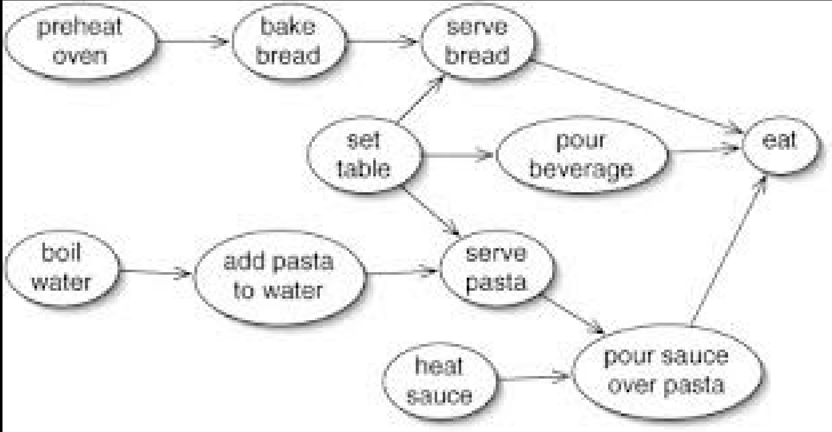

```
Kruskal's algorithm – choose an edge i-j
                      if find(i) != find(j)     //not in same disjoint set
                         union(i, j)   //add the edge, which joins the components
```

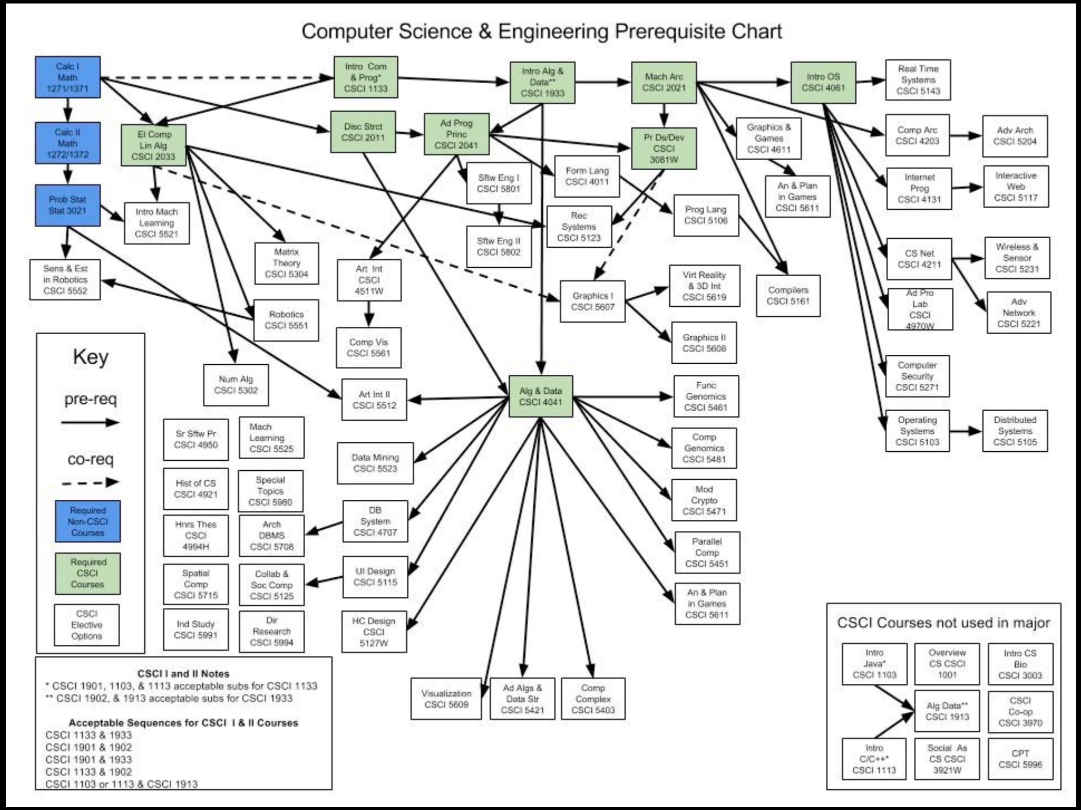
Disjoint sets: {0, 1, 2, 3, 4, 5}

# Topological Sort

# Topological Sort

A topological sort is an ordering of vertices such that if there is an edge from $v_i$ to $v_j$, then $v_j$ comes after $v_i$



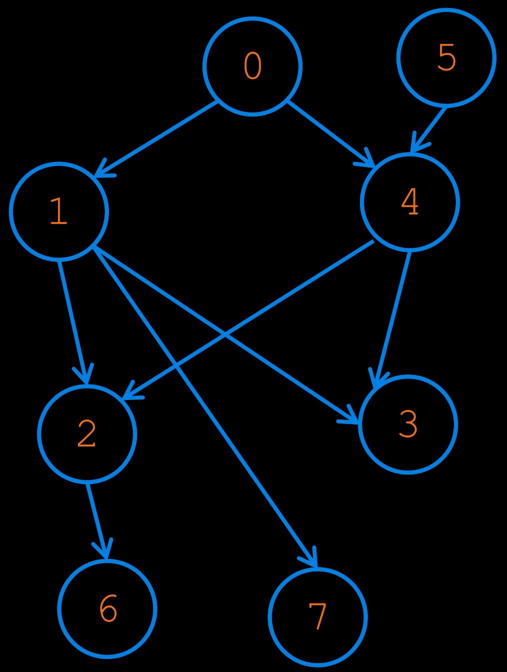



# Topological Sort

A simple algorithm to find a topological ordering is first to find any vertex with no incoming edges.

We can then print this vertex, and remove it, along with its edges, from the graph.

Then we apply this same strategy to the rest of the graph.

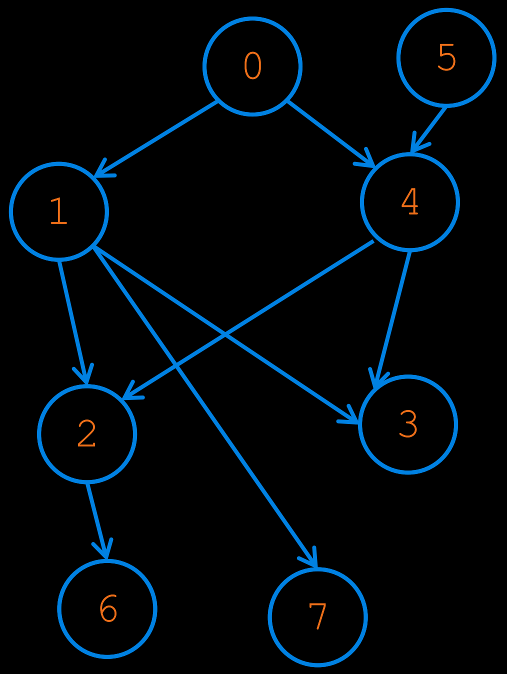


# Topological Sort

A simple algorithm to find a topological ordering is first to find any vertex with no incoming edges.

We can then print this vertex, and remove it, along with its edges, from the graph.

Then we apply this same strategy to the rest of the graph.



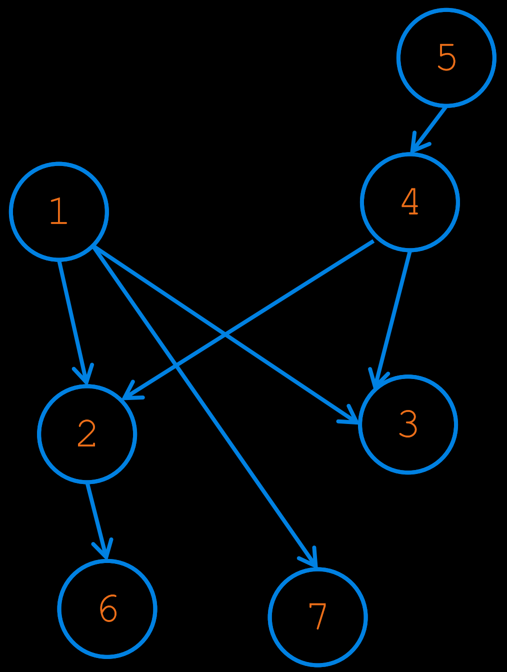

V0 = {0, 5}
Sort Order = {}

# Topological Sort

A simple algorithm to find a topological ordering is first to find any vertex with no incoming edges.

We can then print this vertex, and remove it, along with its edges, from the graph.

Then we apply this same strategy to the rest of the graph.



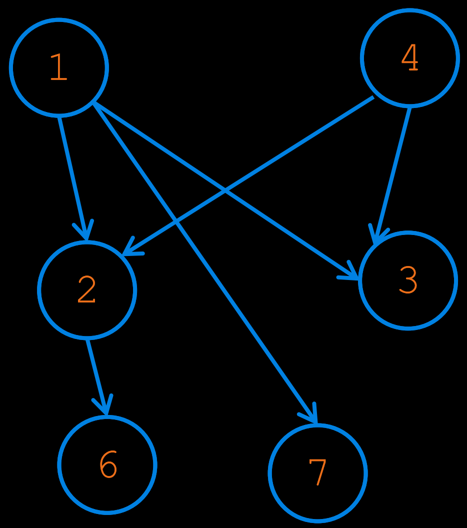

V0 = {5, 1}
Sort Order = {0}

# Topological Sort

A simple algorithm to find a topological ordering is first to find any vertex with no incoming edges.

We can then print this vertex, and remove it, along with its edges, from the graph.

Then we apply this same strategy to the rest of the graph.



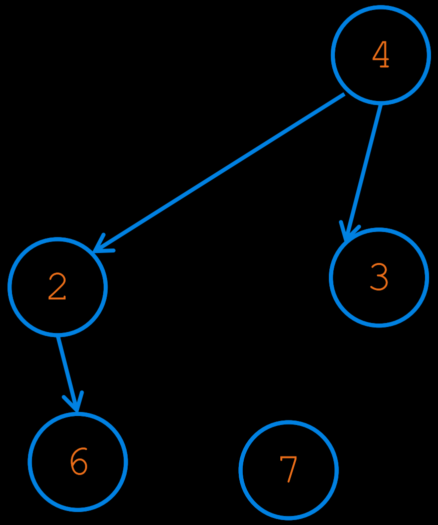

V0 = {1, 4}
Sort Order = {0, 5}

# Topological Sort

A simple algorithm to find a topological ordering is first to find any vertex with no incoming edges.

We can then print this vertex, and remove it, along with its edges, from the graph.

Then we apply this same strategy to the rest of the graph.



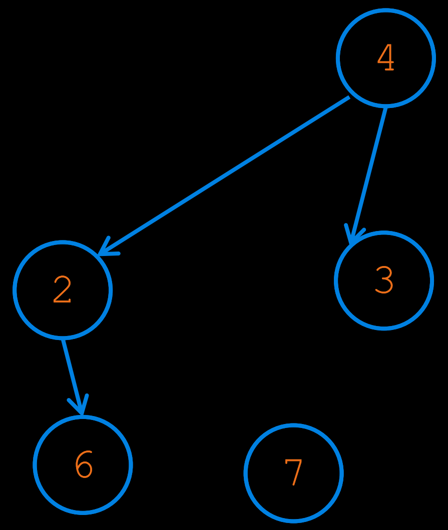

V0 = {4, 7}
Sort Order = {0, 5, 1}

# Topological Sort

A simple algorithm to find a topological ordering is first to find any vertex with no incoming edges.

We can then print this vertex, and remove it, along with its edges, from the graph.

Then we apply this same strategy to the rest of the graph.



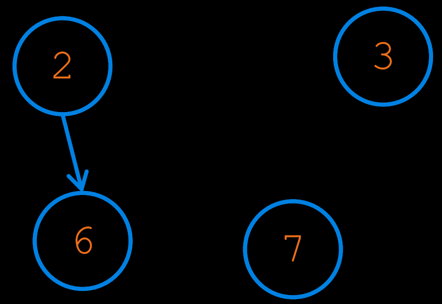

V0 = {4, 7}
Sort Order = {0, 5, 1}

# Topological Sort

A simple algorithm to find a topological ordering is first to find any vertex with no incoming edges.

We can then print this vertex, and remove it, along with its edges, from the graph.

Then we apply this same strategy to the rest of the graph.



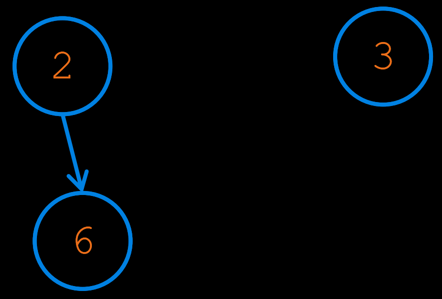

```
V0 = {7, 2, 3}
Sort Order = {0, 5, 1, 4}
```

# Topological Sort

A simple algorithm to find a topological ordering is first to find any vertex with no incoming edges.

We can then print this vertex, and remove it, along with its edges, from the graph.

Then we apply this same strategy to the rest of the graph.




```
V0 = {2, 3}
Sort Order = {0, 5, 1, 4, 7}
```

# Topological Sort

A simple algorithm to find a topological ordering is first to find any vertex with no incoming edges.

We can then print this vertex, and remove it, along with its edges, from the graph.

Then we apply this same strategy to the rest of the graph.


```
V0 = {3, 6}
Sort Order = {0, 5, 1, 4, 7, 2 }
```

# Topological Sort

A simple algorithm to find a topological ordering is first to find any vertex with no incoming edges.

We can then print this vertex, and remove it, along with its edges, from the graph.

Then we apply this same strategy to the rest of the graph.


V0 = {6}
Sort Order = {0, 5, 1, 4, 7, 2, 3 }

# Topological Sort

A simple algorithm to find a topological ordering is first to find any vertex with no incoming edges.

We can then print this vertex, and remove it, along with its edges, from the graph.

Then we apply this same strategy to the rest of the graph.

```
V0 = {}
Sort Order = {0, 5, 1, 4, 7, 2, 3, 6}
```

# Topological Sort Pseudocode

```
void Graph::topsort( )
{
        Queue<Vertex> q;
        int counter = 0;
        q.makeEmpty( );
        for each Vertex v
             if( v.indegree == 0 )
                  q.enqueue( v );
        while( !q.isEmpty( ) )
        {
                Vertex v = q.dequeue( );
                v.topNum = ++counter;  // Assign next number
                for each Vertex w adjacent to v
                       if( --w.indegree == 0 )
                               q.enqueue( w );
        }
        if( counter != NUM_VERTICES )
               throw CycleFoundException{ };
}
```

# Mentimeter

Menti.com

93 47 53 7

# Questions

# Questions